SESSION

PERFORMANCE ISSUES AND ENHANCEMENT METHODS + LOW POWER COMPUTING AND ANALYSIS

Chair(s)

TBA

Reducing Energy Usage of NULL Convention Logic Circuits using NULL Cycle Reduction Combined with Supply Voltage Scaling

Brett Sparkman and Scott C. Smith
Department of Electrical Engineering, University of Arkansas, Fayetteville, AR, U.S.A.
bsparkma@uark.edu and smithsco@uark.edu

Abstract - *The NULL Cycle Reduction (NCR) technique can be used to improve the performance of a NULL Convention Logic (NCL) circuit at the expense of power and area. However, by decreasing the supply voltage, the power of the NCR circuit can be reduced. Since NCR has increased performance, it should be possible to reduce supply voltage to decrease overall power while maintaining the original performance of the circuit. To verify this, the NCR circuit was implemented using a 4-bit by 4-bit dual-rail multiplier as the test circuit, which was simulated in ModelSim to ensure functionality, synthesized into a Verilog netlist using Leonardo, and imported into Cadence to perform transistor-level simulation for power calculations. The supply voltage of the duplicate NCR circuits was decreased until the performance matched the design of the original multiplier, resulting in approximately 25% overall lower energy usage.*

Keywords: asynchronous circuits; NULL Convention Logic (NCL); NULL Cycle Reduction (NCR); supply voltage scaling

I. INTRODUCTION

As circuits are continually produced with increasing numbers of transistors and switching frequencies, circuit power also increases. Although these improvements drastically raise circuit performance, they also have a downside of consuming larger amounts of power, which cause circuits to heat up and last a shorter amount of time on a single battery charge.

This paper demonstrates that overall circuit energy can be reduced for NULL Convention Logic (NCL) circuits [4], while maintaining equivalent performance, by applying the NULL Cycle Reduction (NCR) technique [8] and reducing the supply voltage of the duplicate circuits.

II. PREVIOUS WORK

A. Introduction to NULL Convention Logic

NCL offers a self-timed logic paradigm where control is inherent with each datum. NCL follows the so-called ‘‘weak conditions’’ of Seitz’s delay-insensitive signaling scheme [1]. As with other self-timed logic methods, the NCL paradigm assumes that forks in wires are isochronic [2]. The origins of various aspects of the paradigm, including the NULL (or spacer) logic state from which NCL derives its name, can be traced back to Muller’s work on speed-independent circuits in the 1950s and 1960s [3].

NCL uses symbolic completeness of expression [4] to achieve delay-insensitive behavior. A symbolically complete expression is defined as an expression that only depends on the relationships of the symbols present in the expression without a reference to their time of evaluation. In particular, dual-rail signals or other *Mutually Exclusive Assertion Groups* (MEAGs) can be used to incorporate data and control information into one mixed signal path to eliminate time reference [5]. A dual-rail signal, D, consists of two wires, D^0 and D^1, which may assume any value from the set {DATA0, DATA1, NULL}. The DATA0 state ($D^0 = 1$, $D^1 = 0$) corresponds to a Boolean logic 0, the DATA1 state ($D^0 = 0$, $D^1 = 1$) corresponds to a Boolean logic 1, and the NULL state ($D^0 = 0$, $D^1 = 0$) corresponds to the empty set meaning that the value of D is not yet available. The two rails are mutually exclusive so that both rails can never be asserted simultaneously; this state is defined as an illegal state. Dual-rail signals are space optimal 1-out-of-N delay-insensitive codes requiring two wires per bit.

Most multi-rail delay-insensitive systems [1,4,6], including NCL, have at least two register stages, one at both the input and at the output. Two adjacent register stages interact through their request and acknowledge lines, K_i and K_o, respectively, to prevent the current DATA wavefront from overwriting the previous DATA wavefront, by ensuring that the two DATA wavefronts are always separated by a NULL wavefront.

NCL differs from the other delay-insensitive paradigms [1,6] in that these other paradigms only utilize one type of state-holding gate, the *C-element* [3]. A C-element behaves as follows: when all inputs assume the same value, then the output assumes this value; otherwise the output does not change. On the other hand, all NCL gates are state-holding. Thus, NCL optimization methods can be considered as a subclass of the techniques for developing delay-insensitive circuits using a pre-defined set of more complex components, with built-in *hysteresis* behavior.

NCL uses *threshold gates* for its basic logic elements [7]. The primary type of threshold gate is the TH*mn* gate, where $1 \le m \le n$, as depicted in Fig. 1. TH*mn* gates have n inputs. At least m of the n inputs must be asserted before the output will become asserted. Because NCL threshold gates are designed with hysteresis, all asserted inputs must be de-asserted before the output will be de-asserted. Hysteresis ensures a complete

transition of inputs back to NULL before asserting the output associated with the next wavefront of input data. Therefore, a TH*nn* gate is equivalent to an *n*-input *C*-element and a TH1*n* gate is equivalent to an *n*-input OR gate. In a TH*mn* gate, each of the *n* inputs is connected to the rounded portion of the gate; the output emanates from the pointed end of the gate; and the gate's threshold value, *m*, is written inside of the gate. NCL threshold gates may also include a *reset* input to initialize the output. Resettable gates are denoted by either a *D* or an *N* appearing inside the gate, along with the gate's threshold, referring to the gate being reset to logic 1 or logic 0, respectively.

By employing threshold gates for each logic rail, NCL is able to determine the output status without referencing time. Inputs are partitioned into two separate wavefronts, the NULL wavefront and the DATA wavefront. The NULL wavefront consists of all inputs to a circuit being NULL, while the DATA wavefront refers to all inputs being DATA, some combination of DATA0 and DATA1 for dual-rail inputs. Initially, all circuit elements are reset to the NULL state. First, a DATA wavefront is presented to the circuit. Once all of the outputs of the circuit transition to DATA, the NULL wavefront is presented to the circuit. After all of the outputs of the circuit transition to NULL, the next DATA wavefront is presented to the circuit. This DATA/NULL cycle continues repeatedly. As soon as all outputs of the circuit are DATA, the circuit's result is valid. The NULL wavefront then transitions all of these DATA outputs back to NULL. When the outputs transition back to DATA again, the next output is available. This period is referred to as the DATA-to-DATA cycle time, denoted as T_{DD}, and has an analogous role to the clock period in a synchronous system.

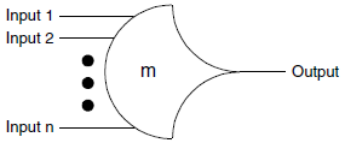


Figure 1. Thmn threshold gate. [8]

The completeness of input criterion [4], which NCL combinational circuits and circuits developed from other delay-insensitive paradigms [1,6] must maintain in order to be delay-insensitive, requires the following criteria: 1. all the outputs of a combinational circuit may not transition from NULL to DATA until all inputs have transitioned from NULL to DATA, and 2. all the outputs of a combinational circuit may not transition from DATA to NULL until all inputs have transitioned from DATA to NULL. In circuits with multiple outputs, it is acceptable, according to Seitz's weak conditions [1], for some of the outputs to transition without having a complete input set present, as long as all outputs cannot transition before all inputs arrive.

Furthermore, circuits must also adhere to the completion-completeness criterion [9], which requires that completion signals only be generated such that no two adjacent DATA wavefronts can interact within any combinational component. This condition is only necessary when the bit-wise completion strategy is used with selective input-incomplete components, since it is inherent when using the full-word completion strategy and when using the bit-wise completion strategy with no input-incomplete components [9].

One more condition must be met to ensure delay-insensitivity for NCL and other delay-insensitive circuits [1,6]. No *orphans* may propagate through a gate [10]. An orphan is defined as a wire that transitions during the current DATA wavefront, but is not used in the determination of the output. Orphans are caused by wire forks and can be neglected through the isochronic fork assumption [2] as long as they are not allowed to cross a gate boundary. This *observability* condition, also referred to as indicatability or stability, ensures that every gate transition is observable at the output, which means that every gate that transitions is necessary to transition at least one of the outputs.

B. Introduction to NULL Cycle Reduction

The NCR technique for reducing the NULL cycle, thus increasing throughput for any delay-insensitive circuit developed according to the paradigms [1,4,6], is shown in Fig. 2. The NCR architecture in Figure 2 is specifically designed for dual-rail circuits utilizing full-word completion, where all bits at the output of a registration stage are conjoined to form one completion signal.

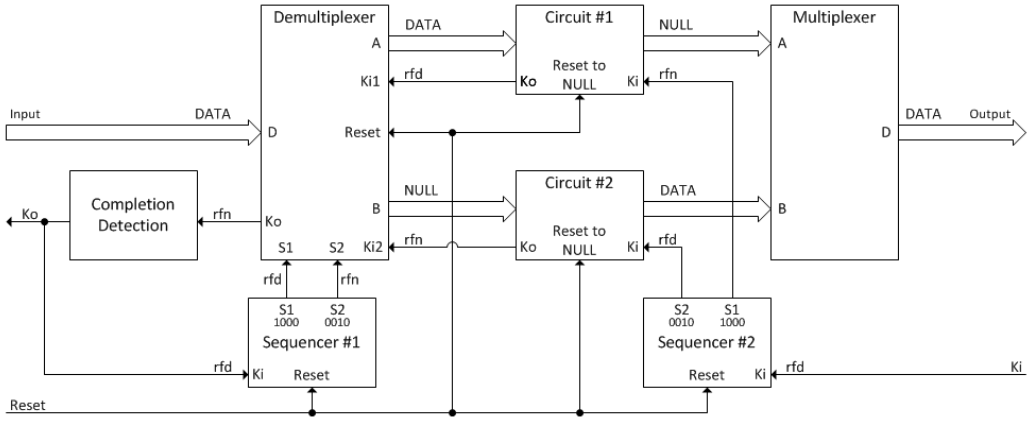


Figure 2. NCR architecture.

Circuit #1 and *Circuit #2* are both dual-rail delay-insensitive combinational circuits utilizing full-word completion, developed from one of the following delay-insensitive paradigms [1,4,6], with at least an input and output registration stage. Additional registration stages may be present, thus further partitioning the combinational circuitry. Both circuits have identical functionality and are both initialized to output NULL and request DATA upon reset. In the case of the NCL paradigm, the combinational functionality can be designed using the Threshold Combinational Reduction method described in [11]; and the resulting circuit can also be pipelined, as described in [12], to further increase throughput. The *Demultiplexer* partitions the input, *D*, into two outputs, *A* and *B*, such that *A* receives the first DATA/NULL cycle and *B* receives the second DATA/NULL cycle. The input continuously alternates between *A* and *B*. The *Completion Detection* circuitry detects when either a complete DATA or NULL wavefront has propagated through the Demultiplexer and requests the next NULL or DATA wavefront, respectively. *Sequencer #1* is controlled by the output of the Completion Detection circuitry and is used to select either output *A* or *B* of the Demultiplexer. Output *A* of the Demultiplexer is input to Circuit #1, when requested by K_{i1}; and output *B* of the Demultiplexer is input to Circuit #2, when requested by K_{i2}. The outputs of Circuit #1 and Circuit #2 are allowed to pass through their respective output registers, as determined by *Sequencer #2*, which is controlled by the external request, K_i. The Multiplexer rejoins the partitioned datapath by passing a DATA input on either *A* or *B* to the output, or asserting NULL on the output when both *A* and *B* are NULL. Figure 2 shows the state of the system when a DATA wavefront is being input before its acknowledge flows through the Completion Detection circuitry, and when a DATA wavefront is being output before it is acknowledged by the receiver.

III. Simulation Results

In order to determine if reducing the supply voltage of a NCR architecture can reduce its power while maintaining performance, a series of simulations was performed. First, a simulation of the VHDL design was performed in ModelSim to ensure that the circuit performed as desired. Next, the files were synthesized using Leonardo in order to generate a Verilog netlist, which was then imported into Cadence. The final steps involved running numerous transistor-level simulations in Cadence using the Analog Design Environment with UltraSim set as the simulator. The results were used to determine the effects of reducing the supply voltage in terms of power and performance.

The parameter names assigned to the multiple supply voltages were as follows: V_{global} for the Demultiplexer, Sequencer #1, and Completion Detection circuitry; V_{local} for Circuit #1 and Circuit #2; V_{mux} for the Multiplexer; and V_{sel} for Sequencer #2. For each simulation using the NCR design, V_{local}, V_{mux}, and V_{sel} were reduced in certain sets. The first voltage reduced was V_{local} because Circuit #1 and Circuit #2 were larger than the Multiplexor and Sequencer #2. Reducing only V_{local} would reduce overall power most effectively. The next voltage reduced was V_{mux} because the Multiplexer was larger than Sequencer #2. The last voltage reduced was V_{sel} because the output select was the smallest out of the three components that the reduced voltage could be applied to. These were reduced until the period and power of the NCR design were lower than that of the single multiplier design, if possible, with a smallest resolution of 10mV.

On the simulation plots, it was noted that the outputs took a short amount of time before they began appearing. This delay occurred because the pipeline took a small amount of time to fill up before the correct output could be observed. In order to calculate the period of the circuit, the period of the main K_o was averaged between the 10^{th} rising edge and the 20^{th} rising edge. Taking the average in this manner ensured that the circuit had reached a steady state. Similarly, the currents of all the voltage supplies were integrated to determine the energy used by the circuit. From this data, the energy per operation was calculated.

A. One-Multiplier Design

Initially, a single 4-bit by 4-bit multiplier, shown in Fig. 3 was chosen as the main circuit. To implement the NCR architecture shown in Figure 2, Circuit #1 and Circuit #2 each consisted of this 4-bit by 4-bit multiplier. The original multiplier design was simulated, and the results are shown in Table I.

TABLE I. ONE-MULTIPLIER DESIGN RESULTS.

Design	V_{global} (V)	I_{global} (µA)	V_{local} (V)	I_{local} (µA)	V_{mux} (V)	I_{mux} (µA)	V_{sel} (V)	I_{sel} (µA)	Period (ns)	Energy / Op (µJ)
One-Multiplier	1.20	121.5							4.18	6.10
NCR Design	1.20	52.38	1.20	153.1	1.20	5.14	1.20	7.19	3.33	8.71
	1.20	45.60	1.10	122.4	1.20	4.62	1.20	6.33	3.76	7.61
	1.20	45.39	1.10	121.6	1.10	4.11	1.20	6.21	3.78	7.56
	1.20	45.48	1.10	120.4	1.10	4.08	1.10	5.48	3.80	7.51
	1.20	39.49	1.00	95.0	1.20	4.30	1.20	5.46	4.36	6.72
	1.20	39.52	1.00	94.7	1.00	3.18	1.20	5.39	4.39	6.67
	1.20	39.26	1.00	92.8	1.00	3.13	1.00	4.23	4.46	6.56

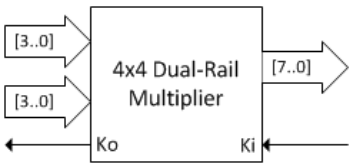


Figure 3. Single 4-bit by 4-bit multiplier.

Unfortunately, there was no possibility of reducing the supply voltages of the NCR design so that the period and power would be less than the single multiplier circuit. The reduction of power with a comparable delay was closest when V_{local} was reduced to 1.00V and everything else remained at 1.2V. The period and power could not be reduced where both would be better than the single multiplier because Circuit #1 and Circuit #2, the multiplier copies, were fairly small; hence, the overhead of the added DEMUX, MUX, and Sequencers outweighed the power savings. If the circuit that was duplicated was larger, a larger power reduction would be seen when compared to the period increase, potentially allowing the circuit to be lower-power and faster.

B. Two-Multiplier Design

To produce a circuit with a lower power and period would require enlarging the duplicated circuit so that reducing the supply voltage would lessen power by a larger amount, needed to compensate for the overhead of the MUX, DEMUX, and Sequencers. Two additional circuits were designed and simulated to test this hypothesis. Although these circuits still used the same multiplier, there were more copies of the multiplier used to generate the larger circuit.

The first additional circuit designed and simulated was two of the multipliers in series, as shown in Fig. 4. The new circuit no longer performed the same function as the original circuit, but it served as a simple example of a larger circuit. The two-multiplier circuit would now take the place of Circuit #1 and Circuit #2 in the NCR architecture. A similar series of simulations was performed.

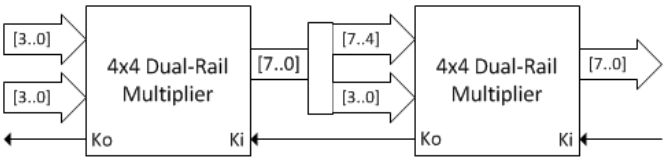


Figure 4. Two-multiplier design.

Using the two-multiplier NCR design, it was possible to achieve lower power while operating slightly faster. The supply voltage parameter settings that accomplished this are shown in boldface type in Table II. Using a 1.04V V_{local} and $V_{mux,}$ while maintaining the 1.2V supply on all other circuit elements, decreased the period by 0.06ns and the energy/operation by 0.57µJ. The decreases correspond to a performance increase of approximately 1.4% and an energy / operation decrease of approximately 4.7%. Although these results were positive, the result was not as beneficial as desired, so another even larger circuit was designed.

C. Four-Multiplier Design

To further demonstrate the advantages of the NCR power reduction technique, an even larger third circuit was designed. This circuit strung together four of the multipliers, as shown in Fig. 5.

Simulating the four-multiplier NCR design further showed the benefits of reducing supply voltages in terms of power. It was possible for the NCR design to consume far less power while maintaining performance. The supply voltage parameter settings that accomplished this are shown in boldface type in Table III. Using a 1.0V V_{local} and V_{mux} while maintaining the 1.2V supply on all other circuit elements decreased the period by 0.01ns and the energy / operation by 6.02µJ. The decreases correspond to a performance increase of approximately 0.2% and an energy / operation decrease of approximately 24.8%. Savings such as this could greatly benefit situations where circuits require lower power to operate. It was observed that the lower MUX supply voltage produced the same lower voltage at the output compared to the original design.

TABLE II. TWO-MULTIPLIER DESIGN RESULTS.

Design	V_{global} (V)	I_{global} (µA)	V_{local} (V)	I_{local} (µA)	V_{mux} (V)	I_{mux} (µA)	V_{sel} (V)	I_{sel} (µA)	Period (ns)	Energy / Op (µJ)
Two-Multiplier	1.20	241.0							4.22	12.21
NCR Design	1.20	51.20	1.20	303.0	1.20	5.11	1.20	6.69	3.39	14.87
	1.20	45.44	1.10	242.4	1.20	4.58	1.20	6.03	3.83	12.77
	1.20	45.59	1.10	241.7	1.10	4.06	1.20	6.03	3.84	12.75
	1.20	45.38	1.10	238.9	1.10	4.01	1.10	5.23	3.86	12.65
	1.20	**41.47**	**1.05**	**212.4**	**1.05**	**3.50**	**1.05**	**4.54**	**4.16**	**11.70**
	1.20	**41.22**	**1.04**	**210.0**	**1.20**	**4.31**	**1.20**	**5.42**	**4.16**	**11.64**
	1.20	**41.20**	**1.04**	**208.2**	**1.04**	**3.44**	**1.20**	**5.44**	**4.19**	**11.56**
	1.20	41.04	1.04	205.3	1.04	3.43	1.04	4.41	4.23	11.46
	1.20	41.26	1.03	203.1	1.20	4.27	1.20	5.36	4.23	11.43
	1.20	41.03	1.03	202.6	1.03	3.33	1.20	5.29	4.25	11.39
	1.20	39.80	1.03	200.9	1.03	3.33	1.03	4.37	4.30	11.30
	1.20	38.85	1.00	187.1	1.20	4.23	1.20	5.16	4.43	10.86
	1.20	39.10	1.00	185.2	1.00	3.11	1.20	5.15	4.46	10.77
	1.20	38.91	1.00	181.7	1.00	3.07	1.00	3.89	4.52	10.65

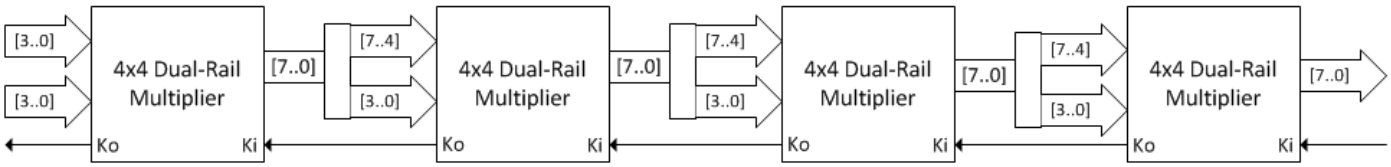


Figure 5. Four-multiplier design.

TABLE III. FOUR-MULTIPLIER DESIGN RESULTS.

Design	V_{global} (V)	I_{global} (µA)	V_{local} (V)	I_{local} (µA)	V_{mux} (V)	I_{mux} (µA)	V_{sel} (V)	I_{sel} (µA)	Period (ns)	Energy / Op (µJ)
Four-Multiplier	1.20	475.1							4.25	24.25
NCR Design	1.20	51.65	1.20	606.4	1.20	5.17	1.20	6.69	3.01	24.23
	1.20	46.12	1.10	485.5	1.20	4.63	1.20	6.04	3.60	21.69
	1.20	45.76	1.10	483.8	1.10	4.12	1.20	6.03	3.61	21.65
	1.20	45.52	1.10	479.8	1.10	4.10	1.10	5.27	3.64	21.60
	1.20	**39.80**	**1.01**	**378.1**	**1.01**	**3.18**	**1.01**	**4.10**	**4.24**	**18.51**
	1.20	**40.19**	**1.00**	**374.8**	**1.20**	**4.26**	**1.20**	**5.16**	**4.21**	**18.29**
	1.20	**39.48**	**1.00**	**372.8**	**1.00**	**3.13**	**1.20**	**5.17**	**4.24**	**18.23**
	1.20	38.94	1.00	366.0	1.00	3.11	1.00	3.93	4.31	18.11
	1.20	39.32	0.99	363.7	1.20	4.32	1.20	5.05	4.29	17.94
	1.20	38.99	0.99	360.9	0.99	3.05	1.20	4.95	4.32	17.85

IV. CONCLUSIONS

Although it was impossible to reduce the power and maintain the performance of the initial one-multiplier NCR design, it was possible to greatly reduce the power while maintaining performance of larger circuits by scaling the supply voltage. By stringing together two-multiplier and four-multiplier NCR designs and performing transistor-level simulations in Cadence to calculate power, it was clearly seen that the power reduction significantly increases as the duplicated circuit size increases. The decrease in power consumption occurred because the lesser supply voltage was distributed over a larger portion of the entire NCR design. The preferred design has a reduced supply voltage connected to only the duplicated circuit; this connection will ensure that the outputs are at the nominal supply voltage level and are therefore equivalent to the original design.

The technique of applying the NCR architecture to a circuit and then reducing the supply voltage to the duplicated circuits could be extremely useful in reducing the power of large circuits. As circuit size increases, the benefits of this technique increase rapidly. The supply voltage levels can be fine-tuned to produce a circuit with the exact same performance as the individual circuit with far less power usage.

REFERENCES

[1] C.L. Seitz, "System timing," in *Introduction to VLSI Systems*, Addison-Wesley, 1980, pp. 218–262.

[2] A.J. Martin, "Programming in VLSI," in *Development in Concurrency and Communication*.: Addison-Wesley, 1990, pp. 1–64.

[3] D.E. Muller, "Asynchronous logics and application to information processing," in *Switching Theory in Space Technology*.: Stanford University Press, 1963, pp. 289–297.

[4] K.M. Fant and S.A. Brandt, "NULL convention logic: a complete and consistent logic for asynchronous digital circuit synthesis," in *International Conference on Application Specific Systems, Architectures, and Processors*, 1996, pp. 261-273.

[5] T. Verhoff, "Delay-insensitive codes—an overview," *Distributed Computing*, vol. 3, pp. 1-8, 1988.

[6] I. David, R. Ginosaur, and M. Yoeli, "An efficient implementation of boolean functions as self-timed circuits," *IEEE Transactions on Computers*, vol. 41, no. 1, pp. 2-10, 1996.

[7] G.E. Sobelman and K.M. Fant, "CMOS circuit design of threshold gates with hysteresis," in *IEEE International Symposium on Circuits and Systems*, vol. II, 1998, pp. 61-65.

[8] S. C. Smith, "Speedup of NULL convention digital circuits using NULL cycle reduction," *Journal of Systems Architecture*, vol. 52, pp. 411-422, 2006.

[9] S.C. Smith, "Completion-completeness for NULL convention digital circuits utilizing the bit-wise completion strategy," in *The 2003 International Conference on VLSI*, 2003, pp. 143-149.

[10] A. Kondratyev, L. Neukom, O. Roig, A. Taubin, and K. Fant, "Checking delay-insensitivity: 10^4 gates and beyond," in *Eighth International Symposium on Asynchronous Circuits and Systems*, 2002, pp. 137-145.

[11] S.C. Smith, R.F. DeMara, J.S. Yuan, D. Ferguson, and D. Lamb, "Optimization of NULL convention self-timed circuits," *Integration, The VLSI Journal*, vol. 37, no. 3, pp. 135-165, 2004.

[12] S.C. Smith, R.F. DeMara, M. Hagedorn, and D. Ferguson, "Delay-insensitive gate-level pipelining," *Integration, The VLSI Journal*, vol. 30, no. 2, pp. 103-131, 2001.

New Single-Phase Adiabatic Logic Family

Mihail Cutitaru, Lee A. Belfore, II
Department of Electrical and Computer Engineering, Old Dominion University, Norfolk, VA, 23529 USA

Abstract— *Power dissipation minimization is the core principle in making any electronic product portable. Even though there has been a decrease in circuit operating voltages, significant power is lost in switching elements (transistors). This has given rise to a new way of computing – adiabatic computing, where significant energy savings are achieved by using time-varying clocks. This paper gives an explanation of how adiabatic computing works and presents several adiabatic families and their advantages and disadvantages. We also propose a new single-phase adiabatic family and show that the proposed buffer/inverter uses 68% less energy than its CMOS equivalent.*

Keywords: Adiabatic logic, low-power computing

1. Introduction

Minimizing power dissipation has been the focus of circuit design for many years, but it has become more important with more new portable devices emerging on the market. As devices get smaller and circuit densities grow, the problem of power dissipation minimization becomes a real concern. Power dissipation is related to the switching activity in transistors and is a function of the square of of the input voltage (V_{dd}^2), so one of the most effective techniques that has been used to decrease the dissipated power is to scale the input voltage. This allows for a reduced power loss, but there will still be a minimum power loss for every erased bit governed by Landauer's Principle. It was shown in [1] that for every erased bit, the lower bound for the energy loss is given by $kTln2$ J, where k is Boltzmann's constant (1.38×10^{-38} J/K), and T is the temperature in degrees Kelvin. Even though this number is not high for a single transistor, modern microprocessors, including the ones used in cell phones and tablets, use hundreds of millions of them. Additionally, current technology consumes almost three orders of magnitude more energy than this limit, so it is important for devices to use the available energy wisely.

One of the ways of completely eliminating power loss is to use reversible logic [2]. Reversible logic is a way of performing calculations so that information that is no longer needed is not dissipated as heat, but saved so that the operations could be undone later. While this is an excellent alternative to using current Complementary Metal-Oxide Semiconductor (CMOS) logic, circuits that take full advantage of reversible logic are still in research stages and few attempts have been made at making circuits more complicated than full adders. Reversible logic is also more difficult to work with since all operations and operands have to be saved and a different architecture is required to accommodate all these changes.

An alternative to CMOS logic is a technique called adiabatic switching, which ideally operates as a reversible thermodynamic process, without loss or gain of energy. Adiabatic computation works by making very small changes in energy levels in circuits sufficiently slow, ideally resulting in no energy dissipation. There are two types of adiabatic computation:

- fully adiabatic — circuit operates arbitrarily slow, loses arbitrarily little energy per operation, and almost all of the input energy is recovered
- partially (quasi) adiabatic — some energy is recovered and some is lost to irreversible, non-adiabatic operations

While fully adiabatic circuits are very attractive from a reversible logic point of view and would be the most suitable for reversible circuits implementation, most proposed adiabatic circuit designs are partially adiabatic because of simplicity and space constraints.

Power loss in conventional CMOS transistors mainly occurs because of device switching and can be easiest understood by studying the CMOS inverter shown in Fig. 1.

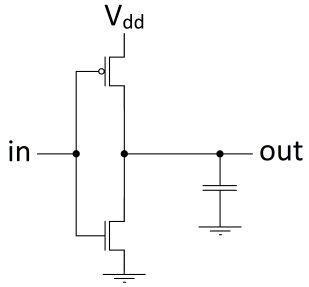


Fig. 1: CMOS Inverter.

An inverter consists of pull-up and pull-down transistors connected to a capacitance C. The capacitance in this case models the fan-out of the output signal. The transistors are in parallel between them and in serial with C. A more compact way to model this is with an ideal switch and a channel resistance R when in saturation mode, as shown in Fig. 2.

When the logic level is set to high, there is a sudden flow of current from the voltage source, through the ideal switch and lumped resistor to the capacitance C, as shown in Fig. 3. The sudden change in voltage level across R accounts for

the large amount of energy lost during CMOS charging.

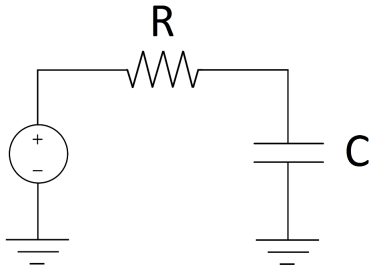


Fig. 2: Equivalent circuit of a CMOS inverter.

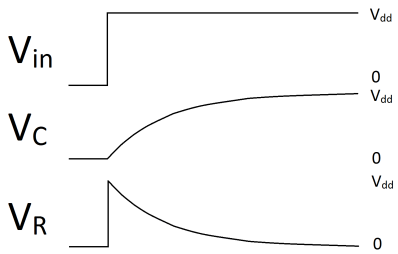


Fig. 3: Voltages at the input, capacitor, and resistor equivalent circuit.

The energy dissipated in this circuit can be modeled by

$$E_{\text{applied}} = CV_{dd}^2 \tag{1}$$

for each clock cycle for a rail-to-rail voltage V_{dd}. When the voltage switches from zero to V_{dd}, the current flow is given by

$$i(t) = \frac{V_{dd}^2}{R} e^{\frac{-t}{RC}} \tag{2}$$

and the power used is

$$P(t) = \frac{V_{dd}^2}{R} e^{\frac{-2t}{RC}}. \tag{3}$$

The amount of energy stored in C is found by integrating the power over time and is given by

$$E_{\text{stored}} = \frac{CV_{dd}^2}{2} \tag{4}$$

which means that half of all supplied energy is stored in the capacitor and the other half is dissipated in R during the charge cycle. In regular CMOS circuits, the energy stored in C is dissipated during the discharge cycle on the falling edge of the clock.

Since the energy stored in the capacitor does not need to be minimized, it is necessary to minimize the energy wasted in the transistor network in order to achieve any energy savings. If the circuit is driven with a frequency f and period T, the total power used in the circuit during a cycle calculated using the above formula yields

$$P_{\text{applied}} = \frac{E_{\text{applied}}}{T} = \frac{CV_{dd}^2}{T}. \tag{5}$$

Adiabatic switching tries to minimize the energy wasted during charging by using a constant current source and charging at a lower frequency f. Both of these optimizations can be calculated by minimizing the function of energy dissipation and current, yielding the dissipated energy formula given by

$$E_{\text{dissipated}} = P\Delta T = \left(\frac{CV_{dd}}{\Delta T}\right)^2 R\Delta T. \tag{6}$$

If charging time ΔT is infinitely long, theoretically there will be no energy dissipated. Infinitely long charging times are impractical, but by spreading out the charge transfer evenly during the charging time, the peak current and large initial power loss are greatly reduced. Adiabatic switching is achieved by replacing the constant DC voltage supply with a time-varying LC driver/oscillator in order to get a constant charging current.

The rest of the paper is organized as follows: Section 2 presents the most commonly used adiabatic oscillators; Section 3 discusses a few of the more important adiabatic logic families and makes a distinction between the ones that use diodes and those that do not; Section 4 describes the proposed adiabatic family and compares its energy usage with CMOS; and Section 5 concludes the paper.

2. Adiabatic Logic Oscillators

A large part of the dissipated energy is lost due to the sudden flow of charge on the rising edge of the square-wave clock, as shown above. Adiabatic computing methods have tried to avoid this loss by making the clock as linear as possible, depending on the design of the logic family. Because the oscillator acts both as a clock and a power supply reference, the convention is to call it a Power Clock (PC).

One of the first proposed adiabatic oscillators has a trapezoidal waveform [7]. This oscillator consists of 4 general stages (Fig. 4): charge, evaluate, discharge, and idle. The output capacitor is charged in the Charge stage, evaluated during the Evaluate stage, and discharged adiabatically back to the PC during the Discharge stage. The PC is then held at ground (GND) during the Idle stage. This type of PC allows the signal to stabilize better during the two plateaus (at V_{dd} and GND), but uses a 4-, 6- or even 8-phase clock, which gets very difficult to control in larger circuits. Additionally, new circuitry is needed in order to generate a linear ramp voltage for charging/discharging.

Another oscillator has a sawtooth waveform. This oscillator allows for a more linear charge of the load capacitance

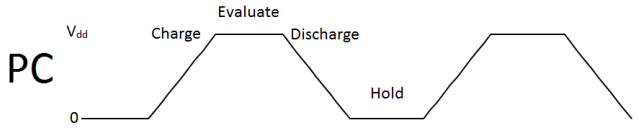


Fig. 4: A single-phase trapezoidal PC with stages marked.

and could perform better than the trapezoidal oscillator since it can have a longer time to charge the outputs at the same frequency. However, it is more difficult to sample the outputs at the peaks since there is no extra time allowed for them to stabilize. A sawtooth oscillator also requires additional circuitry for generation and most logic families using it need to use two phases.

The last type of oscillator has a sinusoidal waveform [9]. This oscillator is a compromise between linear voltage increase and generation circuit complexity. It is easier and more energy-efficient to generate a sinusoidal waveform than a linear waveform, however the sinusoidal waveform does not provide the best approximation for a linear voltage. Most designs with a sinusoidal waveform use a 2-phase clock, although there exist a few designs with a single-phase clock.

3. Adiabatic Logic Families

Adiabatic circuits can be further divided into two categories when it comes to charge-recovery: those that use diodes and those that do not. Using diodes makes the circuit structure simpler, however there is an inherent voltage drop across a diode and the energy dissipated cannot be recovered. The circuits that do not use diodes have better charge-recovery statistics, but are usually larger. Some of the well-known adiabatic families are described below. For the purpose of comparing different adiabatic techniques, each one is shown as an inverter, buffer or both, depending on the logic family.

3.1 Adiabatic families with diodes

1) ADCL
Adiabatic Dynamic CMOS Logic (ADCL) family has a circuit structure that is very similar to a CMOS circuit. The big difference is that the power clock is a single-phase sinusoidal power clock and the GND connection is connected back to the power clock [4], as shown in Fig. 5. The diodes are represented using their pMOS (P1) and nMOS (N2) equivalents. During the Discharge phase, input switches from low to high, PC swings from high to low and the output follows it, thus recovering the charge stored in the capacitor back into the circuit. During the Charge phase, PC swings from low to high, and if the input switches from high to low, the output will follow the PC and charge until PC hits V_{dd}. One disadvantage of ADCL is that the operating speed of the circuit is inversely proportional on the number of gates, so it is not feasible to implement large circuits with this family. An improvement over this family is the Two-Phase drive Adiabatic Dynamic CMOS Logic (2PADCL) family, which has a very similar setup as ADCL [5]. 2PADCL uses 2 complementary power clocks connected at each end of the circuit (instead of 1 clock used in ADCL) and achieves better speeds since the next stage does not have to wait for the output from the current stage to be recovered before computing its output.

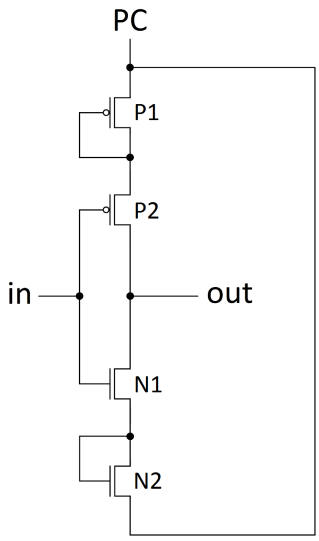


Fig. 5: ADCL inverter [4].

3.2 Adiabatic families without diodes

1) 1n1p
One of the simplest diode-free adiabatic circuits proposed is the 1n1p quasi-adiabatic logic family. This family uses the same setup as a CMOS inverter, but uses a single-phase sinusoidal driver that oscillates between GND and V_{dd} [6]. A diagram of the inverter is given in Fig. 6.

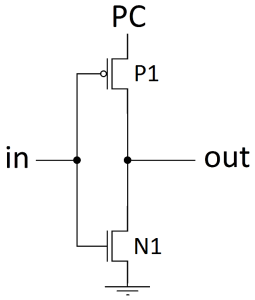


Fig. 6: 1n1p inverter [6].

When the input to the 1n1p inverter is high, the nMOS will be ON and output will be low. When the input

is low, the pMOS will be conducting and the output will charge to high up to the peak of the PC (VPC). As VPC ramps down, the output will follow and the charge stored in the output capacitor will be recovered. This logic family provides good results, however it is not suitable for pipelining. In a pipelined circuit, the next stage of the pipeline should not be affected if its inputs change while it is computing its outputs. In the case of 1n1p logic family, when `out` is connected to to the next stage and the energy stored in `out` starts being recovered, the next stage will not be able to have a constant/reliable output since the input changes. Thus, 1n1p requires the input to be stable for the entire time of computing the output in order to have valid data.

2) 2n2p
This logic family uses 2 nMOS and 2 pMOS transistors to achieve adiabatic operation and compute a logic function and its complement given an input and its complement [7]. The family uses a four-phase trapezoidal clock and the 2n2p buffer/inverter is shown in Fig. 7.

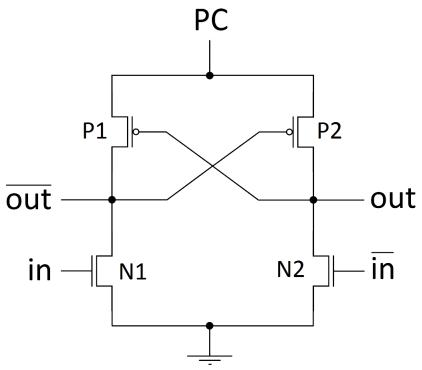


Fig. 7: 2n2p buffer/inverter [7].

The clock begins in the Reset stage where it ramps down. At this time the inputs are low and the outputs are complementary. The output that is high will adiabatically ramp down to low since it is being controlled by the output that is low through a pMOS transistor. During the Wait stage the inputs are evaluated and the corresponding output is assigned a GND values. Since both outputs are at GND during this stage and the upper half of the gate (the 2 pMOS transistors) is held at GND by the power clock, the values of the outputs will not change. In the next stage, Evaluate, the outputs are evaluated based on the resolved inputs. The two outputs will always have complementary values at the end of the Evaluate stage as one is held constant at GND by the nMOS transistor and the other is charged from the PC via one of the pMOSs. The cross-coupled pMOSs guarantee that the two outputs will always be complementary. In the last phase, Hold, the inputs ramp down with the value of PC and the outputs stay constant to be sampled by the next gates.

3) 2n-2n2p
This family is a variation of the 2n2p family and consists of 2 new cross-coupled nMOS transistors added in parallel to the 2 nMOS transistors [7]. The timing and operation of this family is identical to the 2n2p family and the buffer/inverter is shown in Fig. 8. The new nMOS pair make the 2n-2n2p act as a full inverter and the buffer is similar to a Static Random-Access Memory (SRAM) cell. The new nMOSs also have the advantage of eliminating floating nodes in the system, which prevents charge leakage. However, the added transistors prevent it from achieving significant energy savings at speeds above 100 MHz.

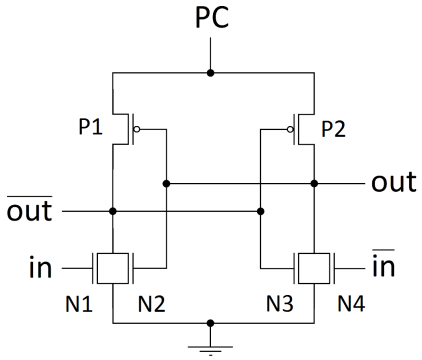


Fig. 8: 2n-2n2p buffer/inverter [7].

4) PAL
Pass-Transistor Adiabatic Logic (PAL) [8] is another variation on the 2n2p logic family described above. In this family, instead of having the lowest node connect to GND, it is connected to the PC, as shown in Fig. 9. This allows for a fully adiabatic operation, but at the cost of higher speeds. Another difference between the two families is that PAL uses a two-phase sinusoidal power clock, which allows for simpler implementation and potentially higher power savings.

5) TSEL
True Single-Phase Energy-Recovery Logic (TSEL) [9] is a partially adiabatic logic family similar to the 2n-2n2p family. This family uses a single-phase sinusoidal power clock with cascades made up of alternating pMOS and nMOS gates. The structure of a buffer/inverter is given in Fig. 10. Each TSEL gate contains a reference voltage (either VRP or VRN, depending on the type of transistor) as a bias voltage that are distinct characteristics of this family. The op-

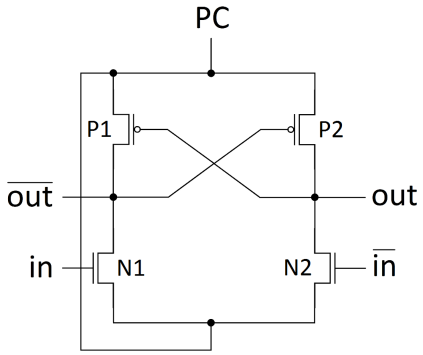


Fig. 9: PAL inverter [8].

eration of a pMOS TSEL gate consists of two stages: discharge and evaluate. During the discharge state the energy stored in the `out` or $\overline{\texttt{out}}$ node is recovered. At the beginning of this stage VPC is high and as it starts ramping down, both outputs are pulled towards the PMOS threshold voltage VTP. This transition is adiabatic until VRP$-$VTP is higher than VPC. During the Evaluation phase, assume that `in` is high and $\overline{\texttt{in}}$ is low. At the beginning of this phase, VPC is low, but as it starts to rise, it turns on P3 and P4, turning on the cross-coupled pMOS transistors P1 and P2. As long as VPC $\leq$ VRP $-$ VTP, P3 and P4 are conducting. Since VRP $>$ VPC, $\overline{\texttt{out}}$ starts rising towards VRP through P4. The two cross-coupled pMOSs (P1 and P2) help amplify the voltage difference between `out` and $\overline{\texttt{out}}$. Once the difference between the output nodes is larger than VTP, P1 turns off and $\overline{\texttt{out}}$ charges adiabatically. When VPC $\geq$ VRP-VTP, P3 and P4 stop conducting and the outputs are disconnected from the lower half of the gate. This allows this family to be immune to any changes that occur in the inputs after P3 and P4 have been turned off. Sampling of the outputs occurs at the end of the Evaluate stage, when the power clock is at its highest.

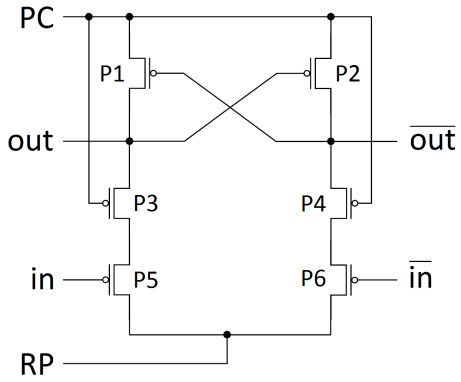


Fig. 10: pMOS TSEL inverter [9].

A nMOS TSEL buffer/inverter operates in a similar way, the only difference being that the stages are reversed from the pMOS TSEL buffer/inverter. The nMOS TSEL gate (Fig. 11) goes through a Charge phase and then an Evaluate phase, and it also uses a different reference voltage VRN. By using a combination of nMOS and pMOS TSEL gates, this logic family is able to achieve very good power savings over other families.

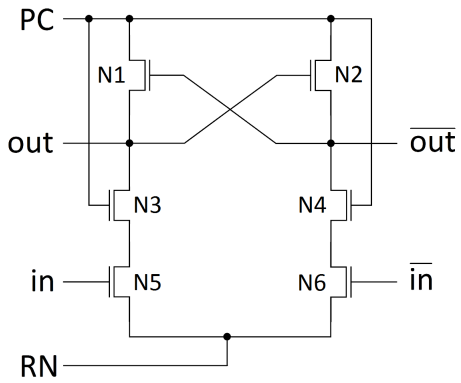


Fig. 11: nMOS TSEL inverter [9].

4. Proposed Design

A new adiabatic logic family is proposed that achieves very good charge recovery and uses a single-phase sinusoidal PC. The proposed buffer/inverter is shown in Fig. 12. When the PC swings from GND to V_{dd}, the value in the `in` input gets assigned to the $\overline{\texttt{out}}$ output and $\overline{\texttt{in}}$ gets assigned to the `out` output, achieving the inverter function. Assuming that `in` is at V_{dd} and $\overline{\texttt{in}}$ is at GND, `in` will cause N3 to turn ON and `out` to be at GND level. When `out` is at GND, P2 is enabled, allowing $\overline{\texttt{out}}$ to ride the PC to V_{dd} level. On the down-cycle, since `out` is at GND level, P1 is enabled, allowing for the energy stored in $\overline{\texttt{out}}$ to be recovered. When the values of the inputs are swapped, the output values are swapped as well.

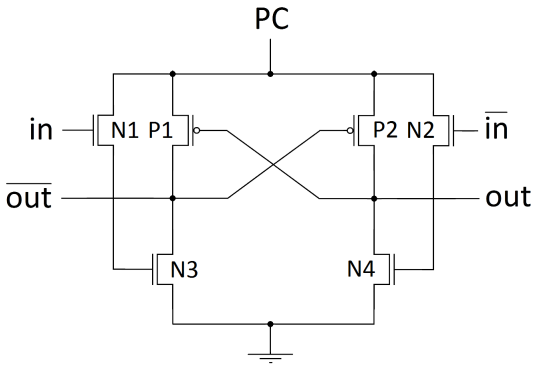


Fig. 12: Proposed Inverter.

The power consumption of the proposed circuit and that of a CMOS circuit were calculated with a SPICE simulation. A 0.25μm process was used with a W/L of 0.36μm/0.24μm respectively and the PC oscillating between 1.8V and GND. The results of the simulation for various frequencies are shown in Fig. 13. As it can be seen from the simulation results, the proposed adiabatic inverter has an energy dissipation of around 68% lower than a CMOS inverter at 100MHz.

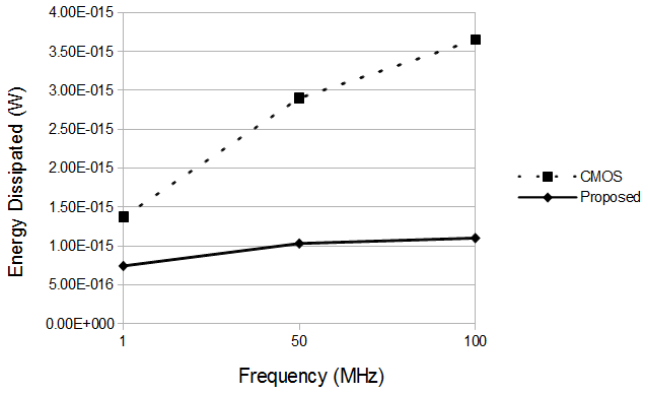


Fig. 13: Energy dissipation comparison between proposed buffer/inverter and CMOS inverter at different frequencies.

The proposed design has the advantage of using only a single-phase PC, which reduces the oscillator complexity, saves on-chip space, and allows for simpler control of connected gates for clock synchronization. The proposed buffer also offers a new way to control the energy flow in the circuit while producing results similar to the ones found in literature [7], [9] by using a single-phase PC.

5. Conclusion and Future Work

This paper explained the basics of adiabatic computation and described the most well-known adiabatic logic families. It was shown that fully adiabatic computation is theoretically possible without any energy loss as the computation times goes to infinity, but for most applications the use of a partially adiabatic logic family is far more suitable. Future work includes the design of larger adiabatic gates and circuits from the proposed buffer/inverter and dissipated energy analysis at higher frequencies and comparison with other adiabatic families.

References

[1] R. Landauer, "Irreversibility and heat generation in the computing process," in *IBM Journal of Research and Development*, Vol. 5, pp. 183-191, July 1961.

[2] C. Bennett, "Logical Reversibility of Computation," in *IBM Journal of Research and Development*, Vol. 17, No. 6, pp. 525-532, Nov. 1973.

[3] S. Younis, "Asymptotically Zero Energy Computing Using Split-Level Charge Recovery Logic," Ph. D. dissertation, EECS, MIT, Cambridge, MA, 1994.

[4] K. Takahashi, M. Mizunuma, "Adiabatic Dynamic CMOS Logic Circuit," in *Electronics and Communications in Japan Part II*, Vol. 83, Issue 5, pp. 50-58, 2000.

[5] Y. Takahashi, et. al., "2PADCL: Two Phase drive Adiabatic Dynamic CMOS Logic," in *Asia Pacific Conference on Circuits and Systems*, pp. 1484-1487, 1996.

[6] V. I. Starosel'skii, "Reversible Logic," in *Mikroelektronika*, Vol. 28, Issue 3, pp. 213-222, 1999.

[7] A. Kramer, et al., "2nd Order Adiabatic Computation with 2n-2p and 2n-2n2p Logic Circuits," in *Proceedings of the 1995 International Symposium on Low Power Design*, pp. 191-196, 1995.

[8] V. G. Oklobdzija, D. Maksimovic, "Pass-transistor Adiabatic Logic using Single Power-clock Supply," in *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, Vol. 44, pp. 842-846, 1997.

[9] S. Kim, M. Papefthymiou, "True Single-Phase Adiabatic Circuitry," in *IEEE Transactions on VLSI Systems*, Vol. 9, No. 1, 2001.

Optimising Energy Management of Mobile Computing Devices

M.J. Johnson and K.A. Hawick
Computer Science, Institute for Information and Mathematical Sciences,
Massey University, North Shore 102-904, Auckland, New Zealand
email: { m.j.johnson, k.a.hawick }@massey.ac.nz
Tel: +64 9 414 0800 Fax: +64 9 441 8181

February 2012

ABSTRACT
Mobile computing devices are becoming ubiquitous and the applications they run are demanding greater processing and storage capabilities. Managing the power consumption and battery life of these devices is increasingly difficult but some careful choices made in the software architecture stack can optimise power utilisation while still maintaining needed services on the architecture. We describe the hardware blocks in a modern mobile device and measure their power requirements. We discuss some power management strategies and present results showing how some quite dramatic energy savings are possible on a typical modern mobile device running Android and Linux. We discuss the implications for future mobile computing device architectures.

KEY WORDS
device architecture; power consumption; battery life; mobile devices

1 Introduction

Modern mobile hardware is extremely complex and a mobile device will typically have more peripherals than a standard desktop PC. The hardware supported by a mobile device usually includes the following: Applications CPU, Baseband CPU, LCD Panel and controller, RAM, NAND flash Memory, MMC flash memory, USB Controller, Audio subsystem, GPS, Video Encoder/Decoder, Serial I/O, Bluetooth, Multiple DSPs, GPU, 2D Graphics Controller, Touchscreen, Cameras, Flashlight, LEDs, Battery Monitor, Wifi. All this hardware is typically powered by a 3.7V battery with a capacity of around 1500mAH. It is vital that the power used by the device is managed efficiently to increase the time that the device is usable [3].

Architects of desktop CPUs have long been aware of power consumption and efforts to produce more power efficient

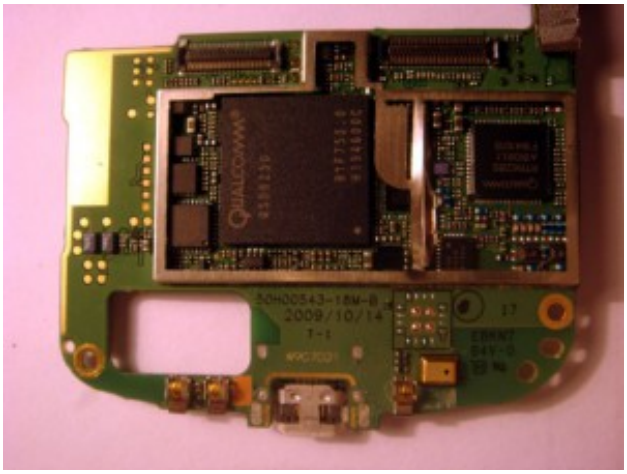


Figure 1: QSD8250 CPU

machines have led to the use of metrics such as FLOPS/watt and MIPS/watt. However, mobile devices have very different requirements and these metrics are not as useful. In 2 we look at the hardware present in a mobile device and discuss it's power management.

Currently, mobile devices make up about fifty percent of all personal computing platforms. This share has increased from almost zero in the last 5 years and mobile devices are becoming ever more pervasive. While it is not vital that a mobile device lasts for weeks on a single battery charge, it is important that it can last a complete day even with heavy use because many users charge their phones overnight every day [1].

Power is measured in Watts, however at a fixed voltage this is proportional to the current in Amps, mobile device battery capacity is always specified in mAH (milliamp hours) so throughout this paper we will us mA as a measure of power consumption. While it is possible to save power by careful design of applications and protocols [8], [9] this paper will concentrate on how the Operating System can be used to manage hardware for best power management.

Future mobile devices will need to use even more power than current devices if they are to support higher speed networking, advanced graphics and distributed resource sharing [2]. Figure 1 shows layout of the main logic board on a typical modern mobile device. Most of the hardware discussed in this paper is integrated into the large ASIC on the left.

2 Hardware Architecture

The hardware in a mobile device consists of a System on a Chip (SoC), memory and a number of peripheral devices. The SoC is an ASIC composed of a number of hardware blocks. The hardware in a modern mobile device will usually include the following blocks.

Application processor: This CPU runs all applications and has access to most hardware, it may be mutlicore. Currently most application processor CPUs are ARM Cortex A8 or A9 designs. The ARM CPU was designed as a low power device and it supports various low power modes which will be discussed in Section 3.3.

Baseband Processor: This processor runs the radio software and controls the radio frequency interface, The baseband processor is either an ARM CPU and DSP or a standalone DSP. Some devices have the baseband integrated in the SoC, others use a separate ASIC. The baseband processor usually runs a proprietary real-time operating system and so it's power management techniques are not available for analysis, however we expect them to be very similar to those used by the application processor.

I2C: A simple 4 wire bus used for controlling slow peripherals such as the touch screen, battery driver and camera CCD. The bus itself has no power management capabilities, power down or sleep commands must be sent to the individual peripherals on the bus.

GPIO: General purpose input/output pins, used for controlling hardware. Each GPIO may have up many functions for controlling internal operation of the SoC and may be multiplexed to SoC I/O pins. There may be many hundreds of GPIOs and each must be configured to draw as little power as possible. A common cause of excess power drain is a misconfigured GPIO.

Display interface: A serial bus used to connect the display panel, common interfaces are MDDI (Mobile Display Digital Interface) and DSI (Display Serial Interface). These buses do include power management capabilities and send messages to the LCD controller about idle states.

Audio: Control of the speaker, handset audio (earcoupled speaker and mic) and headset audio. The audio subsystem can use significant power when playing audio through an amplified speaker.

GPS: Global positioning system, uses a DSP to decode GPS satellite signals for location determination. For fast fix times the mobile network needs to be used to download almanac data. Although the GPS can use significant power it is not always enabled and there are techniques that can be used to minimises it's use [7] [12].

USB: A USB controller in device or host/device mode. The 5V usb power is often used to charge the device so power use is not a issue when the USB port is connected.

SD: Secure digital controller for memory cards, also used for the wifi interface on some devices (SDIO). SD devices use minimal power and may be disconnected and powered down when not is use. Recent Linux kernels have a SD Abstraction Layer (SDAL) which can be used to power down the card when it is idle.

NAND: Flash memory used for storage of the Operating System and data. Recent devices may also have eMMC flash storage. These devices consume very little power.

VFE: Video Front End, used to transfer data from a high resolution camera.

UART: Serial bus connected to the SIM card and Bluetooth controller.

2D graphics engine: Used for block transfers of image data in various formats. May perform scaling and rotation of bitmaps. The 2D engine will often use much less power than the GPU and so unless the 3D engine is needed, graphics composition should use the 2D engine.

DMA: Direct Memory Access, most blocks can transfer data to/from memory without any CPU intervention. This saves power because it allows the CPU to enter a low power state more often.

ADSP: Application Digital Signal Processor, used for processing audio and video data, offload of compute intensive function to the DSP can save power [4].

GPU: Graphics Processing Unit, a 3D graphics engine supporting OpenGL-ES 2.0. The GPU can use significant power, thus the use of user interface features such as complex animated wallpapers should be minimised for best power savings.

MDSP: Modem Digital Signal Processor, used for processing radio frequency signals, this is controlled by the baseband processor.

RAM: Memory external to the SoC, usually Low Power DDR2.

Display: an LCD or LED panel with a resolution of up to 1080x720 pixels. LCD panels use slightly more power than LED panels, modern designs are becoming very power efficient. Until recently, the display was the most power hungry component, this is no longer the case as we show in Section 3.1.

Light sensor: A sensor capable of detecting the ambient light level. This sensor is polled once a second when the display is on and is used to control the backlight for the LCD panel. It uses minimal power and allows the LCD panel to use less power when ambient light is low.

Proximity Sensor: A sensor capable of determinimg if the device is close to another object, this sensor usually uses an Infra-Red emitter and light sensor to detect any reflected light. It uses about 5mA during very short flashes and is only enabled when necessary. It is used to turn off the display when it is placed close to the head during a call. This saves power and avoids the touchscreen being pressed by mistake.

Cameras: A rear facing camera with up to 8Mpixels and capable of streaming HD video. The camera is initialised and focused by sending connads over an I2C bus. There may also be a lower resolution front-facing camera for video calls. Camera use significant power as discussed in Section 3.1.

Accelerometers: Sensors used to detect gravitational or magnetic fields and torque. These sensors do not need to be polled frequently and techniques are available to minimise their effect on power consumption [11].

Real Time Clock: A battery backed clock capable of maintaining the system data and time, it is capable of waking up the device at a set time and must run for long periods of time on minimal power.

Timers: Low and high resolution counters used for system timing. The counters themselves do not use much power but their use should be minimised because they may prevent the device from entering low power modes as discussed in Section 3.3.

Encryption processor: Used for efficient processing of encrypted communication channels. This is controlled by the baseband processor.

Video Processor: Capable of encoding and decoding various video formats including H264 and MPEG4. Software codecs should be avoided because the hardware encoders will use much less power and allow the CPU to idle.

Audio Processor: Capable of encoding and decoding audio formats such as MP3 and AMR. Similarly, these should be used instead of software.

Battery Controller: A device capable of determining the charge remaining in the battery and charging the battery when the device is connected to an external power source. The battery controller usually uses coloumb counting to keep a record of how much power has been used.

Bluetooth: Used for short range RF communications, Bluetooth is designed for low power devices and consequently does not significantly affect power consumption, although some applications may make heavy use of it [10].

Mobile Voice Connectivity: A mobile phone must be connected to the voice network when possible, in areas with good signal strength this can be power efficient, however where the signal strength is low or intermittent, techniques must be used to avoid significant power drain [6].

Mobile Data Connectivity: The mobile data network for a modern device is always connected and so application software must be carefully written to avoid frequent use of mobile data. If possible, synchronisation and other background events should be batched and performed together.

Wifi: Wifi was not designed for mobile devices and so it's power management is more problematic, however modern Wifi adapters use techniques such as Beacon and Idle Mode power saving and Traffic Coalescing [13] to limit the time spent transmitting data. Wifi will generally use less power than the mobile data network and so should be used in preference.

3 Software Architecture

We discuss the software architecture with reference to: an example device; clock control; teh Linux CPU clock control mechanism; and deep sleep issues.

3.1 Example Device

The device we worked with contained a Qualcomm QSD8250 SoC running the Android mobile operating system. Android is an Open Source OS running on top of a

Linux Kernel. For this reason it is easy to modify the software to measure power consumption and see how changes to the software stack affect it. We used a device with a ds2482 battery controller, this is connected via the I2C interface to the SoC and via a proprietary serial interface to the battery.

In order to determine power usage of the device we could replace the battery with a power supply and measure the current drawn but in practice it is easier to use the current measurements provided by the battery controller. The standard software only polls the battery controller once every minute so we had to modify the kernel driver to provide a new interface to the hardware that allowed us to record the current draw at any time. We also modified the driver that controlled charging so that it could be disabled. This allowed us to connect to the device through the USB interface to measure power consumption.

Linux device drivers typically use read or write operations on special files in the /dev directory to communicate with userspace utilities. While we could have used this method, we decided to use a simpler interface, sysfs. Sysfs is a virtual filesystem mounted on /sys, it contains properties of device drivers and can be used to set parameters for a driver.

In Linux, device driver parameters were originally used to allow the kernel boot command line to set up a driver, for example to pass the IRQ number that it should use. Recent Linux kernels allow access to the driver parameters though a set of sysfs files:

```
/sys/module/{driver}/parameters/{param}
```

This interface is very easy to implement and can be also used to perform operations. A macro is used to add parameters to a driver:

```
static int enable_charge=1;
module_param(enable_charge,int,0644);
```

This creates a parameter called enable_charge initialised to 1.

```
module_param_call(battery,set_ch,
                  get_batt,NULL,0644);
```

This creates a parameter called battery, writing to it calls set_ch, reading calls the get_batt function. Modifications to the ds2784 battery driver are shown in Figure 3.1. Example usage of the modified battery driver is:

```
# cd /sys/module/ds2784_battery/parameters
# ls
battery          enable_charge
# echo 0 > enable_charge
# cat battery
```

Hardware	Power Usage
Display	50-80mA
Camera	250mA
GPS	80mA
GPU	90mA
CPU	70-210mA
Bluetooth	15mA*
Wifi	90mA*
3G Data	160mA*
Speaker	80mA
Voice Call	150mA

* only while transferring data

Table 1: Current used by various hardware blocks

```
V=4084696 I=-123950 Iav=-131253 C=1262400
```

Using the modified driver we obtained the power usage values shown in Table 3.1. Note that the camera uses the most power, however this is most likely because it is also heavily using the CPU, GPU and DSP. The average power drawn depends on how the device is being used. When reading text, the CPU will be idle and the device will probably use about 100mA, this gives 15 hours of use with a 1500mAH battery.

The CPU is a significant contribution to the power usage, especially when running at full speed.

Note that when the device is suspended only a very minimal set of systems are left powered on (the always-on subsystem) and we measured a power usage of 2-10mA depending on signal strength and mobile network type.

3.2 Clock Control

The device has a single clock derived from a temperature compensated crystal oscillator (TCXO). This runs at 19.2 MHz or 32KHz when the device is suspended. Other clocks are made using Phase Locked Loop based multipliers (PLLs) and fractional dividers (M/N:D counters) for example on our test device a CPU clock of 998MHz is generated by multiplying TCXO by 52. Four PLLS are available and dividers are used to generate all the other clocks in the system forming a clock tree. There are roughly 150 clocks. Each hardware block uses at least one clock and to save power these clocks must be disabled when not in use. The CPU clock can be modified to slow it down or speed it up, this gives 20 available frequencies between 245MHz and 1.1GHz.

Fiqure 3.2 shows how clock speed affects CPU power usage, as expected this is linear. CMOS transistors use power to switch on and off and so the faster the clock the more power is used. By extrapolating from the data points we can see that even if the clock is switched off, the CPU still

```
static int enable_charge=1;
module_param(enable_charge, int, 0644);

static int battery_adjust_charge_state(struct ds2784_device_info *di) {
....
if(!enable_charge) charge_mode = CHARGE_OFF;
....
}

static int set_charging(const char *val, struct kernel_param *kp) {
        battery_adjust_charge_state(the_di);
        return 0;
}

int get_battery(char *buffer, struct kernel_param *kp) {
        struct battery_status *s=&(the_di->status);
        ds2784_battery_read_status(the_di);
        return sprintf(buffer,"V=%d I=%d Iav=%d C=%d",s->voltage_uV,
                s->current_uA, s->current_avg_uA, s->charge_uAh);
}

module_param_call(battery, set_charging, get_battery, NULL, 0644);
```

Figure 2: Modifications to the ds82482 Battery Driver

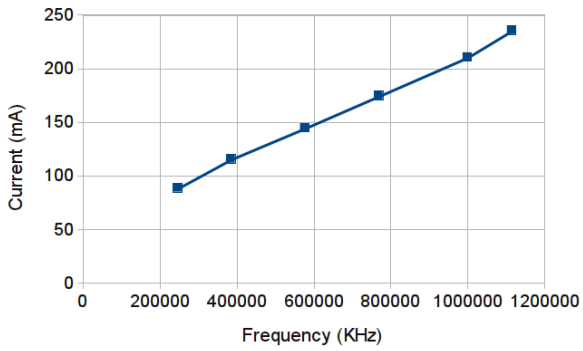


Figure 3: CPU Power usage

draws about 50mA.

3.3 Linux CPU Clock Control

The Linux mechanism for performing CPU clock control is called Cpufreq. This changes the CPU frequency based on load. The CPU clock driver tells the kernel which frequencies are available and provides functions to change frequency. A Cpufreq governor algorithm changes frequency by using various OS load metrics and heuristics. A number of governors are available, the most widely used being the 'ondemand' governor. Cpufreq uses a sysfs interface via file is /sys/devices/system/cpu/cpu0/cpufreq/

Although the CPU clock can not be set to 0 by manipulating the clock hardware, the ARM CPU has a special instruction that can be used for this purpose. The SWFI instruction (Suspend and Wait For Interrupt) disables the CPU clock and enables it again when an interrupt occurs. The Linux idle loop, which runs when no process needs the CPU, executes this instruction. Thus, when the CPU is idle it draws the minimum power in Figure 3.2 (about 50mA). Recent Linux kernels can be 'tickless' this means the kernel doesn't have a periodic timer interrupt so can sleep for long periods of time. Timers can also be 'deferrable' so a non urgent timer interrupt will wait until CPU is not idle. This means that in practice Cpufreq has little effect on power consumption, in fact it can hurt power consumption by preventing it from spending much time idle. The strategy of running as fast as possible and then idling for as long as possible is called 'race to idle'. Whether clock scaling or race to idle is more power efficient depends on a number of factors such as how long it takes to put CPU into an idle state and how memory bound a task is (memory bound tasks are not affected by CPU frequency).

3.4 Deep Sleep

When SWFI is executed, all clocks e.g. timers and bus clocks are still running, it is possible to stop these clocks and save even more power. If this happens then an interrupt can no longer wake the CPU, the only way the CPU can be woken is if the Baseband processor is used to wake it up. This mechanism requires careful cooperation between the application CPU and the baseband CPU.

When we enabled this idle sleep mode the power consump-

tion dropped from 50mA to 30mA.

Even with all the clocks disabled there is still some CPU power usage caused by leakage current. CMOS transistors use power even if they are not clocked and this leakage becomes much worse as the FET channel length is reduced so it is increasingly affecting modern devices.

The solution to minimising this leakage current is to use a global distributed footswitch (or headswitch). This technique involves adding FETs to the power rails of the device (on the V_{dd} rail it is a headswitch, on V_{ss} it is a footswitch) so power can be removed completely. This idle mode is called Deep Sleep or Power Collapse. Once again the baseband processor must wake the application processor but now the application processor must be completely reinitialised.

When we enabled this deep sleep mode the power consumption dropped to 14mA.

The disadvantage of the deeper sleep modes is that they can take a significant time to go to sleep and wake up. Thus, they should only be used when an interrupt is not expected shortly. For asynchronous interrupts, it is not possible to know if one about to happen, however the most common interrupt is a timer interrupt and the OS can determine when this will next happen. Thus, the idle code can look at when the next timer interrupt is due and choose an appropriate sleep mechanism. There are also other constraints on sleep modes, for example on the device we were using, GPU interrupts could not be used to wake from either of the deep sleep modes, this may explain why the measured power consumption of the GPU is relatively high.

4 Voltage Control

The power used by a CMOS device is proportional to the clock frequency, but it is also proportional to the square of the voltage. This means that lowering the CPU voltage should save more power than lowering the frequency. For a given frequency there is a minimum V_{dd} at which the CPU will function, this voltage also depends on temperature because the gate delay of a CMOS transistor increases with temperature.

4.1 Static Voltage Scaling

Static Voltage Scaling (SVS) is the technique of using a set CPU voltage for each possible CPU frequency. This voltage must be chosen such that the CPU will still function at all temperatures.

In order to implement SVS, an extra field is added to the CPU frequency table to specify a voltage for that frequency. It is important that the voltage is changed at the correct time, the voltage must be increased before increasing the frequency but decreased after decreasing the frequency. This ensures that the voltage is always at or above the required value.

4.2 Dynamic Voltage Scaling

The voltage used for SVS must be chosen so that it is safe for all devices and thus it has to be chosen very conservatively. It would be better if the device could provide feedback about the voltage. This is possible if some extra hardware is included, this is called Dynamic Voltage Scaling (DVS) [5].

The SoC we were using has hardware to support DVS, it uses ring oscillators and a number of delay circuits to model the longest possible delay through three different parts of the CPU, the Datapath, the Floating Point Unit and the Level 2 Cache. We can use these delay circuits to tell if V_{dd} is too high or too low. If any oscillator thinks V_{dd} is too low, it must be increased, if all think it is too high, it can be decreased. Since the delay depends on temperature, the die temperature must also be measured and taken into account. Although the device we were using supported DVS it was not implemented in the Linux kernel, we implemented a device driver for DVS.

Our driver uses a 2D table of voltages for frequencies and temperatures, it polls the hardware every 200ms until the voltage for the current temperature has stabilised. When the voltage for a particular temperature has stabilised, the table value is fixed so no more polling is necessary. Module parameters are used for status and to enable/disable DVS. When a voltage is set, the table is updated to clamp to this voltage for all lower frequencies and temperatures.

Output from the status of the driver is shown below.

```
# cat status
Current TEMPR=9
Current Index=1
Current Vdd=925
Vdd Table:
 0:   19200 925 0
 1: 245760 925 0
 2: 384000 925 0
 3: 576000 975 0
 4: 768000 1150 1
 5: 998400 1225 1
 6:1113600 1300 0
```

This shows that frequencies of 768MHz have stabilised with voltages of 1.15V and 1.225V respectively.

Figure 4 shows the affect of SVS and DVS on power consumption. With a fixed V_{dd}, this voltage must be chosen for the highest possible frequency, thus the device uses significantly more power at lower frequencies. The values for the SVS driver are provided by the Google kernel for this

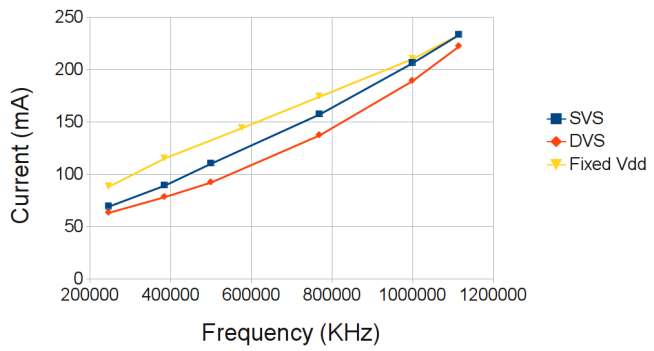


Figure 4: Voltage Scaling Results

device, it looks like the values have been determined by finding the best voltages for the highest and lowest frequencies and drawing a straight line between them. This means that DVS only provides significant power savings for frequencies in the middle of the table.

5 Conclusions

We have investigated various mechanisms for saving power on a mobile device including clock control, deep sleep and voltage control. In summary, we found that deep sleep (power collapse) provides the greatest power savings but it is not always available because there is significant overhead involved in putting the device into this mode. Changing clock frequencies can achieve significant power savings but only for tasks that are not CPU intensive.

Voltage scaling helps save power and has the potential to save more power than frequency scaling because of the quadratic relationship between voltage and power. Dynamic voltage scaling is a useful technique but if the voltages for static voltage scaling are chosen correctly is does not provide significant power savings.

In the future we would like to extend this work by investigating other devices and mechanisms for reducing power usage. Thes e ideas are likely to be of importance for tablet computers as well as for mobile phones.

References

[1] Ferreira, D., Day, A.K., Kostakos, V.: Understanding human-smartphone concerns: a study of battery life. In: Proc. 9th Int. Conf. on Pervasive Computing (Pervasive'11). pp. 19–33. No. 6696 in LNCS (2011)

[2] Furthmuller, J., Waldhorst, O.P.: Energy-aware sharing with mobile devices. In: Proc. Eighth Int. Conf. on Wireless On-Demand Network Systems and Services. pp. 52–59 (2011)

[3] Gordon, D., Sigg, S., Ding, Y., Beigl, M.: Using prediction to conserve energy in recognition on mobile devices. In: Proc. IEEE Int. Conf. on Pervasive Computing and Communications Workshops (PERCOM Workshops). pp. 364–367. Seattle, WA, USA. (21-25 March 2011)

[4] Hameed, R., Qader, W., Wachs, M., Azizi, O., Solomatnikov, A., Lee, B.C., Richardson, S., Kozyrakis, C., Horowitz, M.: Understanding sources of inefficiency in general-purpose chips. In: Proc. 37th Int. Symp. on Computer Architecture (2010)

[5] Hormann, L.B., Glatz, P.M., Steger, C., Weiss, R.: Evaluation of component-aware dynamic voltage scaling for mobile devices and wireless sensor networks. In: Proc. IEEE Int. Symp on World of Wireless, Mobile and Multimedia Networks (WoWMoM). pp. 1–9. Lucca, Italy (20-24 June 2011)

[6] IEEE: IEEE Draft Standard Methods for Measuring Transmission Performance of Analog and Digital Telephone Sets, Handsets, and Headsets - Amendment 1. IEEE P269a/D5.2, January 2012 pp. 1 –33 (19 2012), 6135531

[7] Kjaergaard, M.B., Bhattacharya, S., Blunck, H., Nurmi, P.: Energy-efficient trajectory tracking for mobile devices. In: Proc. 9th Int. Conf. on Mobile Systems, Applications and Services (MobiSys'11). Bethesda, MD, USA (28 June - 1 July 2011)

[8] Liu, Y., Guo, L., Li, F., Chen, S.: An empirical evaluation of battery power consumption for streaming data transmission to mobile devices. In: Proc. ACM Multimedia Conference (MM'11). Scottsdale, AZ, USA (28 November - 1 December 2011)

[9] Meng, L., Shiu, D., Yeh, P., Chen, K., Lo, H.: Low power consumption solutions for mobile instant messaging. IEEE Trans. Mobile Computing PP99, Online, 1–18 (June 2011)

[10] Park, U., Heidemann, J.: Data muling with mobile phones for sensornets. In: Proc. 9th ACM Conf. on Embedded Networked Sensor Systems (2011)

[11] Priyantha, B., Lymberopoulos, D., Liu, J.: Littlerock: Enabling energy-efficient continuous sensing on mobile phones. Pervasive Computing April-June, 12–15 (2011)

[12] Thiagarajan, A., Ravinranath, L., Balakrishnan, H., Madden, S., Girod, L.: Accurate, low-energy trajectory mapping for mobile devices. In: Proc. 8th USENIX conference on Networked systems design and implementation (NSDI'11). Usenix Association, Berkeley, Boston, MA, USA (30 March - 1 April 2011)

[13] Wang, R., Tsai, J., Maciocco, C., Tai, T.Y.C., Wu, J.: Reducing power consumption for mobile platforms via adaptive traffic coalescing. IEEE Journal on Selected Areas in Communications 29(8), 1618–1629 (September 2011)

Design and Low Power Implementation of a Reorder Buffer

J.D. Fisher, C. Romo, E. John, W. Lin
Department of Electrical and Computer Engineering,
University of Texas at San Antonio
One UTSA Circle, San Antonio, Texas, U. S. A.

Abstract - *Superscalar microprocessor designs demand high performance with limited increased power overhead. To obtain high throughput rates out-of-order execution is implemented, along with the use of dispatch buffers and reorder buffers (ROBs). ROBs can become high power consuming components that require power aware designs. This research investigates different design techniques of the ROB with emphasis on power efficiency. Mainly, the centralized and distributed ROB designs were implemented and power analysis was completed. The distributed ROB was determined to be more power efficient by 12.84%. This distributed system truly embraces the concept that efficiencies can be gained just by taking a process and optimizing it for parallelism and resource sharing that will ultimately lead to an overall optimization such as power consumption.*

Keywords: Reorder Buffer, Microarchitecture, Power Efficient

1. Introduction

A standard in the development of today's technology is to achieve the execution of multiple instructions through instruction level parallelism (ILP). ILP is most efficiently exploited through the use of superscalar processors. Out-of-order execution achieves an increased performance in superscalar microprocessors through the utilization of a reorder buffer (ROB) to complete instructions in program order to the architected register file. Although highly ambitious, the overall objective of the ROB is to obtain a performance increase equal to that of the paralleled instructions.

The implementation of an ROB can result in a fairly high increase in power consumption. This performance increasing component should therefore be designed to minimize all design tradeoffs including power overhead. As microprocessors migrate to smaller technologies, power becomes more and more of an issue that needs to be addressed. Such is the purpose of analyzing components in the overall design such as the ROB. It is important that no component be a power consumer unless it is doing so efficiently. Ultimately, high performance processors need to be power aware in their design in order to provide reliable long lasting products.

This research focuses on the following theories as a foundation for designing a power efficient ROB:

- the centralized ROB architecture [1] and [2],
- the distributed ROB architecture [3],
- the ability to utilize the concept of dynamic voltage scaling in order to break the ROB into pieces and provide power only to sections of the ROB that need the power [4],
- utilizing fast comparators that would focus the power mainly when a match is found [5],
- dynamic ROB resizing [6].

Mainly, the centralized and the distributed ROB are investigated and implemented in this research to demonstrate the ability to minimize power in a microprocessor. The rest of this paper is organized as follows: Section 2 provides an overview on power consumption, superscalar pipelining and the ROB, Section 3 describes related work, Section 4 provides the details of our design and implementation, Section 5describes the methodology used for accomplishing the research, Section 6 presents the results and analysis, and Section 7 summarizes our research in a conclusion.

2. Background

2.1 CMOS Power Dissipation

Power dissipation analysis is a key component in the progression of new technology. Evaluation of power involves two power dissipations, static and dynamic [7]. The most significant sources of dynamic power dissipation are the charging and discharging of capacitance (switching) and short circuit power dissipation. The techniques to reduce dynamic power include reducing the switching voltage. But since voltage thresholds do not scale equal to the supply voltage, performance loss is experienced [7]. Parasitic capacitance and switching frequency can also be reduced; however application of these requires detailed understanding of the design as to

minimize the tradeoffs involved. In this research, reduction of the switching frequency is focused on circuit optimization, through alternate logic implementation. Its emphasis is on reducing power dissipation by reducing the switching frequency, all the while maintaining functionality and performance.

Static power is related to the logical states of a circuit that equate to leakage current. There are two dominate types of leakage current addressed in this research, reverse PN-junction and subthreshold channel current [7]. There are several techniques available for reducing leakage power which can include, transistor stacking, power gating, multi-threshold voltage, etc.

2.2 Superscalar Pipelining

The basis of pipelining requires an instruction to follow a hardware flow that starts with the fetching of the instruction and finishes with the results written to the architected register file. By dividing the flow into smaller stages throughput can be increased [8]. Jouppi's classification of the pipelined process is understood through four stages: instruction fetch (IF), instruction decode (ID), execution (EX), and write back (WB) [8]. From a resource perspective each stage independently performs its function allowing it to run operate in parallel with the other stages.

Superscalar pipelined microprocessors are designed to fetch and issue multiple instructions every machine cycle, meaning the stages as well as the instructions can be executed in parallel [8]. The major hindrance is encountered through stalls in the execution stage of the pipeline. To address this draw back the execution stage is divided into parallel execution units, known as a diversified pipeline. In this setting the instructions are dispatched through a buffer to the appropriate execution unit and executed out-of-order. This requires a mechanism, such as the ROB, to write back instructions in program order to avoid validity issues. This is also known as a dynamic superscalar pipelining [8]. With each additional level of parallelism the complexity of the design also increases.

2.3 Reorder Buffer

A ROB is a vital component in microprocessors that allows for program order instructions to be issued in parallel, based on the instruction-level parallelism, execute them out-of-order, and completed them in-order. ROB designs are most often implemented as a circular buffer utilizing the FIFO queue data structure. The general concepts of operation of the ROB are given in Figure 1.

The critical aspect of the ROB is maintaining knowledge of the program order. The first step is to allocate entries in the ROB according the program order during the initial dispatch of the instruction. The tail of the buffer is then incremented accordingly. In addition, corresponding data is also allocated in the ROB such as the destination registers, a destination register tag created for the matching of completed instructions to its corresponding ROB entry, and a result field entry is created. The ROB then waits for the instructions to execute and assigns the resulting data to the appropriate program instruction result field. Once this is accomplished a ready flag indicates that the instruction is ready for completion. The final step completes the instructions and increments the head of the ROB. The head instruction must be completed in order to maintain the program order. The general operations of the ROB are the foundation to performing out-of-order operations in a superscalar design, but implementation can be completed through various approaches.

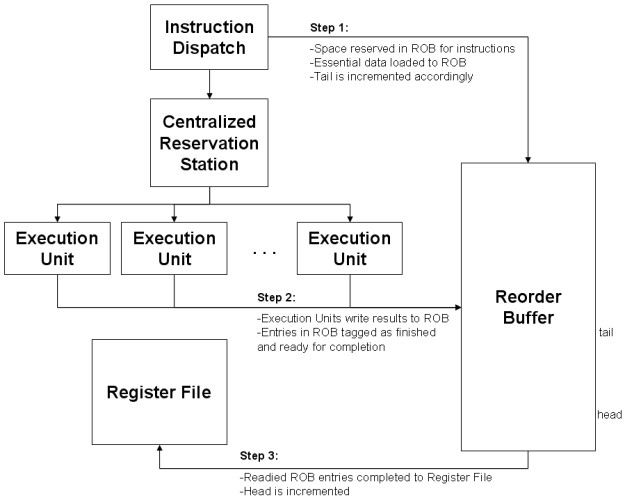


Figure 1. Generalized ROB

3. Related Work

There are many ROB designs, each with specific emphasis that are integral to the foundation of the trends to theory, design, and implementation of the ROB. This chapter takes into account the designs that were investigated and describes the work which was revealed.

The design of the centralized ROB is the culmination of research done by Lennel et. al. [1] and Wallace et. al. [2]. The research provides a detailed analysis of the functionality, theoretical implementation, and interaction of the ROB with its neighboring components. The investigation by Kocuk et. al. [3] provides the distributed ROB layout in theory without emphasis on how that implementation strategy can be realized. The strategy was heavily motivated to optimize power. The centralized and distributed ROB designs independently provide the details in which this research was founded because the designs are different enough in architecture, but similar enough in

functionality that the designs could be easily compared to provide a power enhancement case.

Wann-Yun et. al. investigated the low power techniques through dynamic voltage scaling (DVS) [4]. Applying such techniques to a FIFO queue with only a subset of entries in use could significantly relieve the unnecessary consumption of power. Overhead must be considered in the design to avoid performance degradation [4]. Implementation strategy would be based on the partitioning of the ROB into to subsets regulated by a DVS controller. An algorithm is created to determine the timing that can be placed on the entries to make them idle for the wait state of the stages flow [4]. The key aspect of dynamic voltage scaling is to ensure that the overall design will not be impacted negatively by the hardware intensive strategy that makes up dynamic voltage scaling and its controllers.

Kucuk et. al. addressed several issues in the design of the ROB including the investigation of a power efficient design of comparator [5]. A large ROB is a highly intensive read and write component that can have a slow and inefficient comparator scheme when attempting to match destination tags. This research also investigates the approach of dissecting the current ROB design and applying zero byte encoding to the design to alleviate power usage in the ROB.

The investigation completed by Ghose et. al. attempted to segment a large ROB and dynamically resize it depending on trends as instructions are flowing [6]. This is the basis for dynamic ROB resizing and if implemented correctly can work much like DVS without significant overhead [6]. The crucial element is the algorithm to determine when to power down a partition and when to power up the partitions in order to have them available when needed without interrupting throughput [6].

4. Design and Implementation

Two ROB designs were implemented to investigate approaches in providing a power efficient ROB component. The two designs, the centralized and distributed ROB models, are compared based on power consumption, disregarding external forces. The designs were implemented using Verilog to the specification of a Spartan 3E FPGA. By utilizing the FPGA it is possible to compare the two designs without attention to technology or other varying constraints. The Spartan 3E FPGA is a 90nm technology chip that utilizes a 50 MHz clock and each design utilized Xilinx ISE® in order to map the design to the pins on the FPGA.

Implementation on an FPGA required that the clock be sliced into four phases. The first phase (0) of the clock sets the internal registers that will be able to assist with further processing, such as the tail + 4 register that will assist in allocating the four entries in the ROB. The second phase (1) of the clock performs all output signals. The reservation station output for the tag to go with the instructions and the register file output that provides the destination register and the results. If results are truly completed then this stage will also increment the head to the appropriate instruction in program order. The third phase (2) of the clock is the point at which entries are allocated from dispatch and elements are populated to accommodate the new instructions. The fourth phase (3) implements the final stage in which the ROB is written to by the execution units with their results based on the tag that was sent previously in second phase.

Design assumptions include an instruction-level parallelism of four, 32 entries in the ROB, 16 usable registers, and each register has a value up to 16 bits. These assumptions allow for the designs to build on the same foundation to allow for a one-to-one comparison.

4.1 Centralized ROB

The centralized ROB design is based on a circular buffer that is represented by a first-in-first-out queue with a head and tail pointer to showcase the entries being worked on. The entries outside the range of the head and tail are stale data that will be overwritten as the instructions flow through the design as new instructions are added to the queue. The ROB is broken into 32 entries with various data elements for processing. Figure 2 provides a graphical explanation to the elements described and the bit locations for each element in the ROB entries. The specific data elements that of the 32 entries are described in Table 1.

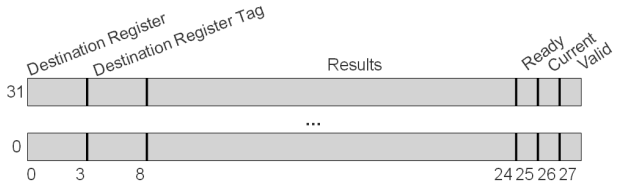


Figure 2. Centralized ROB elements

Table 1. Centralized ROB Elements

Element Name	#Bits	Description
Destination Register	4	The final destination register the results will be written to. The 4 bits allow for 16 possible registers.
Destination Register Tag	5	Tag generated for register renaming. The 5 bits allow for the 32 ROB entries.
Data	16	The actual results that will be written to the appropriate destination register.
Current	1	Flag to indicate the most current destination register entry in the ROB
Ready	1	Flag used to indicate when an entry is ready to be completed.
Valid	1	Flag used to indicate if the value is valid.

The data path of the centralized ROB is illustrated in Figure 3. The flow follows the previously described 3 step strategy. Specifically the first step is to allocate the entries in the ROB for the four new instructions being dispatched in parallel. At dispatch the ROB creates four entries, adds four to the tail pointer, adds a destination register for each entry, generates the destination register tag for each entry, updates the current bits to one for this destination register entry and updates all other entries for the same destination register to zero, sets the ready bit to zero, and sets the valid indicator at one. The ROB then sends the tags associated to each entry to the appropriate reservations station to join up with the entry that holds the source registers sent from the dispatch. This function is vital because it allows the tag to flow with the instruction through execution and will act as the key to get the results in the correct ROB entry.

Once the instructions are executed and results are available the execution units will use the generated tag as to write the results to the ROB and update the ready bit to one. The ROB can complete an instruction only if the head instruction is ready, indicated by a ready bit of one. The entry is then completed as a write to the register file for that particular destination register. The head is then incremented for each completed instruction in order to keep the queue referencing the appropriate entries.

It is important to note that this centralized ROB design is one large I/O buffer. Large amounts of data are held in this one large buffer and therefore there is a heavy burden when data is written to or read from this buffer. The searching can become tedious for the execution units as they try to write their results to the buffer, but need to ensure they are on the right entry. These are some of the concerns that lead this design to showcase a non-power efficient design, and the implementation will showcase that as such.

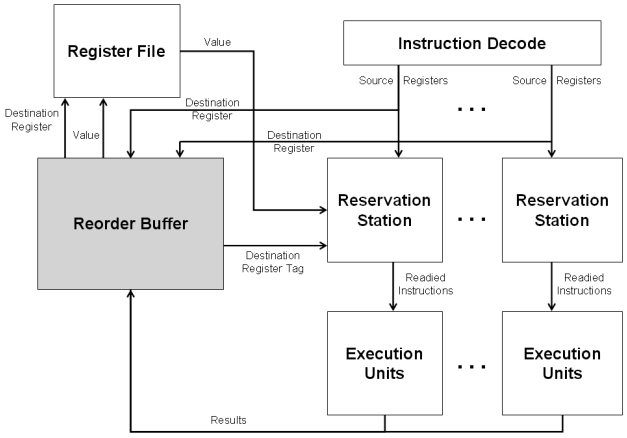


Figure 3. Centralized ROB

4.2 Distributed ROB

The design of the distributed ROB has one centralized ROB component that holds pointers to several distributed ROB components, 8 components, specifically for this design. The distributed ROB is broken into 32 entries with various data elements for processing. Figure 4 provides a graphical explanation to the elements described in the centralized ROB component and the bit locations for each element in the ROB entries. The specific data elements that of the 32 entries are described in Table 2.

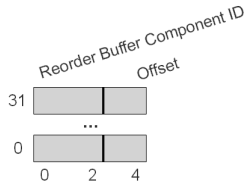


Figure 4. Centralized ROB elements of Distributed ROB

Table 2. Centralized ROB elements for Distributed ROB

Element Name	#Bits	Description
ROB Component ID	3	The pointer indicating in which distributed ROB component the entry lives, 3 bits allow for 8 different distributed ROB components.
ROB Component Offset	2	The pointer indicating in which of the 4 entries within the distributed ROB components the entry lives, 2 bits allow for 4 entries to exist in each ROB component.

The 8 distributed components of the distributed ROB contain 4 entries each with various data elements for processing. Figure 5 provides a graphical explanation of the elements described in the distributed ROB components and the bit locations for each element. The specific elements of the 4 entries are described in Table 3.

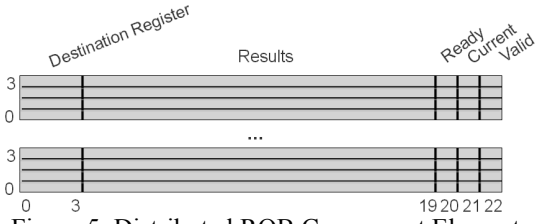


Figure 5. Distributed ROB Component Elements

Table 3. Distributed ROB Component Elements

Element Name	#Bits	Description
Destination Register	4	final destination register the results will be written to, 4 bits allow for 16 registers
Data	16	The actual results that will be written to the appropriate destination register.
Current	1	Flag to indicate the most current destination register entry in the ROB.
Ready	1	Flag used to indicate when an entry is ready to be completed.
Valid	1	Flag used to indicate if the value is valid.

The datapath of the distributed ROB is depicted in Figure 6. Much like the centralized ROB, the flow follows the 3 step strategy previously described. The difference is that the tag now points to a specific ROB component and the offset to determine which of the four entries in the ROB component the tag is associated with. The allocation scheme is that it is done in a round robin format. Therefore, the first instruction would allocate to the first ROB component and its first offset and then the next would go to the second ROB component and its first offset. This round robin format is important to the possible utilization enhancements from a read/write port perspective.

Once the instructions are executed and results are available the execution units will use the generated tag as the key and write the results to the ROB components and update the ready bit to one. It is at this point the ROB components are ready to complete an instruction if the instruction that is now readied is the head.

The ROB components checks if their entries are the head to see if that entry has a ready bit of one and if it is, the entry is then completed as a write to the register file for that particular destination register. Once that is complete, the head is incremented for each completed instruction that has been completed in order to keep the queue referencing the appropriate entries.

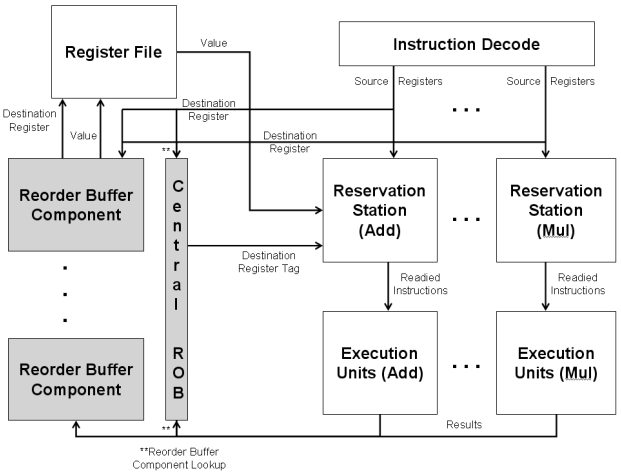


Figure 6. Distributed ROB

5. Methodology

The Verilog designs were imported and compiled using Xilinx ISE®. To validate the semantics of the Verilog code, a test bench was created which provides a program snippet to showcase the behavior of the circuit as a whole. The snippet showcased six clock cycles and utilized all four stages of the clock in each clock cycle. The design functioned properly in behavioral simulation, as well as the Post Place & Route simulation for the building, mapping, and timing of the design, and determining the pins associated to on the Spartan 3E FPGA. The last function of the Post Place & Route simulation was the creation of the XPower Analyzer netlist used for the dynamic power analysis. Loading the netlist into the XPower Analyzer determined the overall power consumption and provided the data for the power analysis.

6. Results

6.1 Behavioral and Post Place & Route Simulation

The simulations of the validity of the design provided one set of results for evaluation. To verify the functionality of the each design, the Verilog HDL code was placed through simulations which ultimately determine the validity at various checkpoints. The signals used in the validation simulations are described in Table 4.

Table 4. Simulation Elements

Element Name	#Bits	Description
oPhase	2	clock phase that will cycle from 0-3 & continue to represent all phases in the simulation.
iDestReg	16	destination register value that is sent from the instruction dispatch, used to allocate the ROB entry
iDestRegTag	20	destination register tag which is generated and used for register renaming purposes within the ROB entries, value is passed with the data from the execution unit to be able to update the correct ROB entry
iData	64	value from the execution units that is to be written back to the correct ROB entry.
oDestReg_rf	16	Output of the ROB that tells the register file which registers to update
oData_rf	64	data that is being sent to the register file to be updated for the specific registers.

Although the designs were different, the output signals which send completed data to the register file remained the same based on the similar clock cycle phases. This was the emphasis of the simulation results which proves the validity of the data. The behavioral and Post Place & Route simulations yielded the same results concluding that each design was functionally implemented on the FPGA according the design strategy described previously in Section 4.

6.2 Power Analysis

Power analysis provided the second set of results for evaluation. Each design and its corresponding netlist were loaded into the XPower Analyzer application. For the purpose of the designs mentioned in this research the input data is received from the instruction decode stage as well as the execution units. The output is the register results that

are written back to the register file at the end of a program instruction's cycle.

The Vcc power of the Spartan is divided into three different components: the Vcc(int), Vcc(aux), and Vcc(out). The first investigated was on the dynamic power consumption. Figure 7 specifies the two designs by dynamic power consumption. The Vcc(int) shows a difference of 23.64% on power consumption due to dynamic power based on the design being distributed. Each design has the same amount of Vcc(out) dynamic power consumption. Lastly, the Vcc(aux) is not impacted by dynamic power.

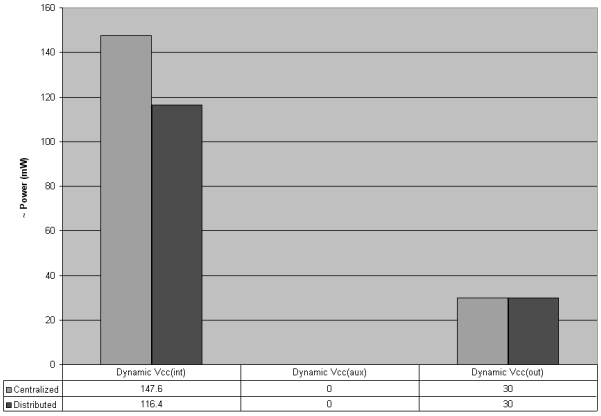


Figure 7. Dynamic Power Distribution

In order to understand the 23.64% difference in dynamic power for Vcc(int), the various components and how they were impacted by the different designs were examined. In Figure 8 the specific components to the FPGA and their impacts on power consumption are shown. Note that the signals component on the FPGA has a difference of 66.67% from the centralized to the distributed design. The signals component on the FPGA indicates the power consumed by data signals and control signals throughout the design based on the testbench's netlist. It can be seen that each design remains the same in clock power and logic power; therefore it is the signal power driving the efficiencies in the design.

With dynamic power as such a focus on the design and implementation of the ROBs it is reasonable that the designs showcase a 23.64% increase in power efficiency from the centralized ROB design to the distributed ROB design. To take it more granular that 23.64% is possible because of the 66.67% increase in efficiency for power consumption of the signals, data and control, that make up the power consumption of the Vcc(int).

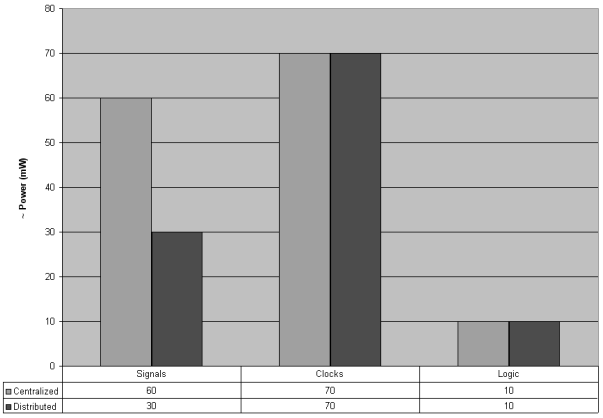


Figure 8. Power Consumption of Specific Components

Next, static power was investigated. Static power consumption is indicated the Vcc (int, aux, and out). Figure 9 indicates that there is not much of a change for the centralized and distributed designs. The Vcc(int) shows a difference of .064% while the Vcc(aux) and Vcc(out) remain constant across the two designs. The focus of Figure 8 is to showcase all the static power of the Spartan 3E FPGA due to the implementation on the chip.

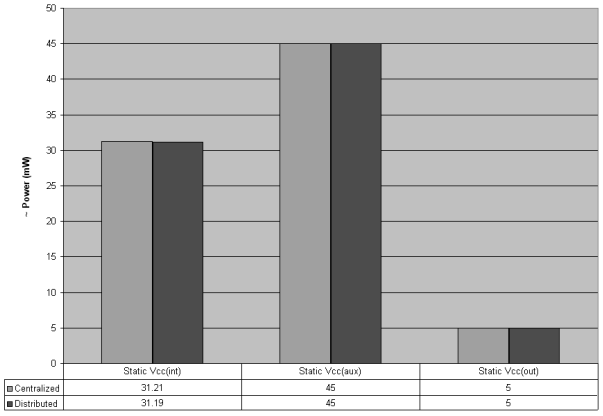


Figure 9.Static Power Distribution

The static power results for these designs were expected. The static power of the design indicates the leakage power of the Spartan 3E FPGA. Since both the centralized and distributed ROBs were not designed to be more efficient from a sizing scale, each design holds the same number of elements. Also to mention that since the design utilizes similar components on the Spartan 3E FPGA it would mean that the leakage to have each of those components would remain the same.

The final results examine the total power consumption of the designs. Figure 10 provides the static, dynamic, and overall power consumption. The total static power of the designs have a difference of .024% based on the Vcc(int) being the only static component that had a change from centralized to distributed. The total dynamic power of the

designs have a difference of 19.26% based on the Vcc(int) being the only dynamic power consumer that changed from centralized to distributed. Lastly the overall power consumption of the designs have a difference of 12.84%.

Ultimately the results conclude that the distributed ROB is a more efficient design from a power consumption perspective. These results are based on both designs being implemented on the same Spartan 3E FPGA device with the same testbench to benchmark the designs.

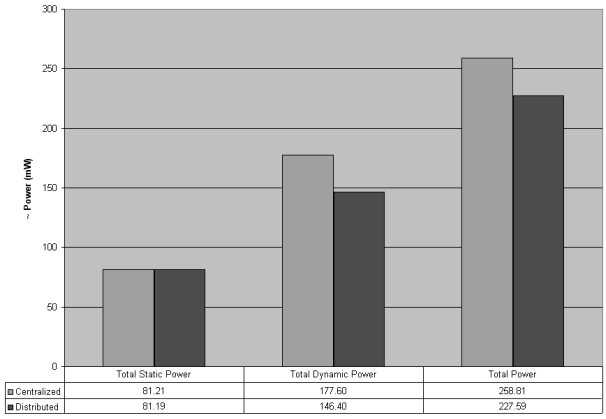


Figure 10. Total Power Consumption

The design aspects of the distributed ROB that make it more power efficient are the smaller and less logic intensive centralized ROB component and the distributed ROB components. The centralized ROB component acts as merely a lookup table that points the instruction dispatch and the execution units to the correct ROB component where the actual entry exists. Without the need to constantly read and write large bits of data to this centralized buffer the logic decreases significantly and power efficiency is gained.

Since this centralized buffer in the design is made of a smaller number of elements comprising each entry, this buffer is much smaller in scale to the normal centralized ROB design. The smaller buffers allow for significantly less intensive logic to perform reads, writes, and eliminate searches. With the logic focusing on smaller partitions of the overall ROB it allows for a stronger degree of parallelism within the ROB logic. The distributed ROB design focus on resource sharing, allows for the logic to provide more power efficiently. The second key to the efficiencies of the ROB components is in the manner of instruction decode. When the first instruction makes an entry in the ROB component the next instruction will jump to the next ROB component to load to. In doing so there is only need for one port to send data to the register file since there are eight ROB components and the instruction parallelism is four. Port eliminations is an important concept that can drive power consumption to be more efficient.

7. Conclusion

The ROB is a vital component of dynamic superscalar microprocessors to allow for out-of-order processing of instructions to increase throughput. Although the ROB is vital to the superscalar microprocessor it can be a large hindrance due to overall power consumption and efficiencies due to different architectural styles. Based on the power consumption issues, the technique of designing the ROB in a more dynamic power efficient manner was in focus.

In order to understand the ROB and possible power efficiency strategies the two main designs, centralized and distributed, were analyzed in detail and designed into strategies that could be realized. The realization of the two designs was with Verilog and implemented according to the Spartan 3E FPGA, which allows for an in depth picture of power consumption from both a static and dynamic perspective.

The final results showcase that the overall distributed ROB is more power efficient than the centralized ROB by a power reduction of 12.84%. Overall, this research provided a deeper look into the ROB, provided insights on how to design the ROB more power efficient in an effort to make the overall microprocessor more power efficient, and provided an implementation strategy that was able to be validated against.

8. References

[1] J. Lenell, S. Wallace, N. Bagherzadeh, "A 20MHz CMOS ROB for a Superscalar Microprocessor," *4th Annual NASA VLSI Symposium*, November 1992.

[2] S. Wallace, N. Dagli, N. Bagherzadeh, "Design and Implementation of a 100 MHz ROB", *37th Midwest Symposium on Circuits and Systems*, August 1994.

[3] G. Kucuk, O. Ergin, D. Ponomarev, K. Ghose, "Distributed ROB schemes for low power," 21st *International Conference on Computer Design*, October 2003.

[4] S. Wann-Yun, C. Hsin-Dar, "Saving register-file static power by monitoring short-lived temporary-values in ROB," *13th Asia-Pacific Computer Systems Architecture Conference*, August 2008.

[5] G. Kucuk, K. Ghose, D. Ponomarev, P. Kogge, "Energy-Efficient Instruction Dispatch Buffer Design for Superscalar Processors", *International Symposium on Low Power Electronics and Design*, August 2001.

[6] K. Ghose, D. Ponomarev, G. Kanad, "Energy-Efficient Design of the ROB", *International Workshop on Power and Timing, Modeling, Optimization and Simulation*, Sept 2002.

[7] Gary K. Yeap, "Practical Low Power VLSI Design," Kluwer Academic Publishers, 1998.

[8] John Paul Shen and Mikko H. Lipasti, "Modern Processor Design: Fundamentals of Superscalar Processors", McGraw Hill, 2005.

SESSION

ALGORITHMS, LOGIC, CIRCUIT/HARDWARE DESIGN, AND TOOLS

Chair(s)

TBA

Area-Time Efficient Digit-Serial-Serial Two's Complement Multiplier

Essam Elsayed and Hatem M. El-Boghdadi

Computer Engineering Department, Cairo University, Egypt

Abstract *- Multiplication is an important primitive operation used in many applications. Although parallel multipliers produce results fast, they occupy considerable chip area. For applications with lengthy operands such as cryptography, the required area grows further. A Digit-Serial-Serial multiplier receives both inputs serially one digit per cycle. This reduces area at the expense of the number of cycles required to complete the multiplication. Digit multiplier designs are flexible with respect to the digit width offering designers the opportunity to select the most suitable compromise between area and cycle count for the application in concern.*

In this paper, a new Digit-Serial-Serial multiplier is proposed that is efficient in terms of area and area-time product. The proposed multiplier supports one operand to be of dynamic-width while the other operand is fixed-width. In contrast, other multipliers support only fixed, equal-width operands. With a small modification, the multiplier is shown to be able to operate on 2's complement operands. The proposed multiplier also supports bit-level pipelining. That is, independent of the operand width and the digit width, the critical path of the multiplier pipeline stage can be reduced down to the delay of a D-FF, an AND gate and two full adders (FAs) independent of the digit width.

Simulation results show that the proposed multiplier reduces the required area over similar multipliers [1] by up to 20% and reduces the area-time product by up to 32%.

Keywords: Multiplier, Digit-Serial-Serial, Area, Time, 2's complement, dynamic operand

1 Introduction

Multiplication is a core operation in hardware designs. Although many multiplier designs have been proposed in literature, these designs still have a room for improvement [7].

To speed up the multiplication operation, completely parallel implementations have been proposed [3][9]. Since these designs process the two operands in parallel they possess a short processing time. In this case, parallel systems require a considerably large silicon area and area reduction becomes essential.

Reduction of multiplier area has been the field of study of many papers [1]. Many approaches were followed to reduce the area. One of the common approaches used to reduce area is to split multiplication over multiple cycles and re-use a smaller circuit that exploits the similarities in the multiplication operation. This requires splitting one or the two operands into a number of digits each of width *d* bits and processing one digit at a time. Basically, multiplication can be carried out one digit at a time producing one digit of the result per cycle. This approach has many variations.

One variation is Digit-Serial-Parallel multiplication [7] where one operand is input in parallel while the other operand is input one digit per cycle. This has the advantage of reducing the multiplier area but suffers from the fact that one of the operands is still handled in parallel keeping the area-required high.

Another variation is the Bit-Serial-Serial multiplication where both operands are input one bit per cycle [8]. This has the advantage of minimal area but suffers from high cycle count.

The third is Digit-Serial-Serial where both operands are input one digit at a time filling the gap between the two former approaches. In this paper, we follow this approach and propose an area and area-time product efficient multiplier.

Many design approaches are used to achieve Digit multipliers. Two systematic approaches are folding [10] and unfolding [7]. Folding starts with a full parallel multiplier and truncates the basic execution unit (in the multiplier case, the adder) to the digit width rather than the operand width. On the other hand, unfolding starts with a Bit-Serial-Serial multiplier and replicates it a number of times equal to the digit width. Other systematic approaches are based on radix 2^n arithmetic [1]. An ad-hoc approach also can be found in [7] producing a Digit-Serial-Parallel multiplier. Our design adopted the folding approach starting from a fully parallel multiplier and truncating it down to a Digit-Serial-Serial multiplier.

Compared to the large volume of research on Digit-Serial-Parallel multipliers, a much smaller volume is present for Digit-Serial-Serial multipliers. Perhaps one of the first trials to achieve Digit-Serial-Serial multipliers is the work proposed by Aggoun *et al.* [1]. They proposed a design that is based on radix-2^n arithmetic and supports bit-level-pipelining. This is achieved by using 4-to-2 compressors [1] instead of adders. Their proposed multiplier is considered the first Digit-Serial-Serial multiplier in literature and accordingly is compared to Digit-Serial-Parallel multipliers. Though their multiplier was shown to outperform the compared-to multipliers in area and area-time product measures, however they impose some constraints like no 2's complement support, no odd digit count support and no dynamic-width operand support.

Almiladi [2] made an extension to the work of Aggoun [1] where he used another design methodology (Table Methodology) that is more systematic to achieve the same results and accordingly their multiplier inherits the same constraints. Also their multiplier does not support bit-level-pipelining and introduces extra startup cycles. That is, the result digits are not produced immediately starting from the first cycle as in Aggoun [1] or the multiplier proposed in this paper.

Up to our knowledge, this paper is the third trial to address Digit-Serial-Serial multipliers. In this paper, a new Digit-Serial-Serial multiplier is proposed that is efficient in terms of area and area-time product. The area-time product is a balanced measure that considers both the speed and the area factors of a design rather than considering each alone. We call the proposed multiplier the Folded Digit-Serial-Serial Multiplier (FDSSM).

Let the two operands to be multiplied be *A* and *B* of width *m* bits and *n* bits respectively. In the FDSSM operation, *A* and *B* are divided into digits of width *d* and fed serially; one digit of each of the operands per cycle. The FDSSM is shown to require $(n + m)/d$ cycles to complete the multiplication operation.

The FDSSM supports one operand of dynamic-width while the other operand (the smaller of the two operands) is of a fixed-width. In contrast, other functionally-similar multipliers support only fixed, equal-width operands. With a simple modification, the FDSSM can also perform 2's complement operands multiplication that is not supported by the other functionally-similar multipliers.

The FDSSM also supports bit-level pipelining. That is, when the multiplier is pipelined; the pipeline stage delay is independent of the operand width and the digit width. In the FDSSM case; it can be pipelined with a pipeline stage delay down to the delay of one D-FF, one AND gate and two FAs.

The main reason the FDSSM improves the area is the elimination of the input buffers and the last-digits' buffers; simulation results show that the FDSSM reduces the required area over other functionally-similar multipliers by up to 20%. It also reduces the area-time product by up to 32%.

The next section describes the mathematical background of the Digit-Serial-Serial multiplier. In section 3, we describe the FDSSM circuit and its operation. In section 4, we present the simulation results and a comparison to other multiplier designs. Section 5 covers the FDSSM's properties such as dynamic operand support and how the FDSSM can be modified to support 2's complement and applying bit-level-pipelining. Finally, Section 6 presents the concluding remarks.

2 Mathematical Background

In this section, we present the mathematical background that is used as the base of the FDSSM. In the following we consider the two multiplied operands to be of the same width. Later in section 5 we show how one of the operands can be of dynamic width.

Consider the multiplication of two numbers *A* and *B* each of width *n* bits where $A=a_{n-1}a_{n-2}\ldots a_0$ and $B=b_{n-1}b_{n-2}\ldots b_0$. The paper and pencil technique to calculate $A \times B$ has two main stages; the partial product bits generation where each bit of *A* is multiplied by each other bit of *B*. The second stage is to accumulate the partial product bits of the same significance to form the final result. The partial product bits' matrix (PPBM) to be accumulated can be given as follows:

$$\begin{array}{ccccccccc} \underline{col\ \ 2n-2} & & \underline{col\ \ n} & \underline{col\ \ n-1} & \cdots & \underline{col\ \ 1} & \underline{col\ \ 0} \\ & & & a_{n-1}b_0 & \cdots & a_1b_0 & a_0b_0 \\ & & a_{n-1}b_1 & a_{n-2}b_1 & \cdots & a_0b_1 & \\ & & & \vdots & & & \\ a_{n-1}b_{n-1} & \cdots & a_1b_{n-1} & a_0b_{n-1} & & & \end{array} \quad (1)$$

Column i of the matrix is of significance 2^i where $0 \le i \le 2n-2$. Accumulating the partial product bits of each column gives the multiplication result bit of the current column's significance and multiple carry bits for the next significant column.

The operand *A* can be appended with zero-bits at both the least and most significant ends to make the PPBM a rectangular matrix. This extension can be done as follows:

$$a_i = 0 \quad \forall i : n-1 < i \le 2n-1 \ or\ i < 0 \quad (2)$$

The resulting PPBM with effective partial product bits underlined (non-underlined terms are equal to 0) is:

$$\begin{array}{cccccccc} a_{2n-1}b_0 & a_{2n-2}b_0 & \cdots & a_nb_0 & \underline{a_{n-1}b_0} & \cdots & \underline{a_1b_0} & \underline{a_0b_0} \\ a_{2n-2}b_1 & a_{2n-3}b_1 & \cdots & \underline{a_{n-1}b_1} & \underline{a_{n-2}b_1} & \cdots & \underline{a_0b_1} & a_{-1}b_1 \\ & \vdots & & & & & & \\ a_nb_{n-1} & \underline{a_{n-1}b_{n-1}} & \cdots & \underline{a_1b_{n-1}} & \underline{a_0b_{n-1}} & \cdots & a_{-n+2}b_{n-1} & a_{-n+1}b_{n-1} \end{array} \quad (3)$$

After generating the PPBM, the second stage is to accumulate these partial products. This can be formulated as:

$$A \times B = \sum_{i=0}^{2n-1} \sum_{j=0}^{n-1} a_{i-j} b_j 2^i \quad (4)$$

The outer summation represents the columns of PPBM and the inner one represents the rows.

Each cycle, the FDSSM accepts one digit of *A* and one digit of *B*. Grouping each *d* consecutive bits of *A* into a digit, equation (4) can be written as:

$$A \times B = \sum_{i=0}^{(2n/d)-1} \sum_{j=0}^{n-1} \sum_{l=0}^{d-1} a_{id+l-j} b_j 2^{id+l} \quad (5)$$

Grouping each d consecutive bits of B into a digit the equation becomes:

$$A \times B = \sum_{i=0}^{(2n/d)-1} \sum_{j=0}^{(n/d)-1} \sum_{l=0}^{d-1} \sum_{k=0}^{d-1} a_{d(i-j)+(l-k)} b_{jd+k} 2^{id+l} \qquad (6)$$

Each of the outer summation iterations represents d consecutive columns (column set) of the PPBM of equation (3). Within the FDSSM; each of these iterations is processed in one cycle starting by the least significant column set requiring a total of $2n/d$ cycles to complete the multiplication.

The number of partial product bits generated per cycle is calculated as the product of the number of iterations of the three remaining inner summations of equation 6, $(n/d) \times d \times d = nd$ bits. Also the partial product digits count per cycle is calculated as the partial product bits count per cycle nd divided by digit width d to be n digits. The partial product digits are then accumulated in the same cycle using a tree of 4-to-2 compressors of depth (log n) *-1* levels and one normal adder.

3 The Proposed Multiplier

As mentioned in the previous section; the FDSSM runs over two stages per cycle: partial product digits generation and partial product digits accumulation. In this section, we propose an implementation for the FDSSM.

3.1 Partial Product Digits Generation

Generally speaking, the main idea is to divide multiplication into identical sets of operations that can be processed by a smaller processing unit over multiple cycles. This is in contrast to the multiplication as a single set of operations processed by a full-length processing unit in one cycle. This is shown in Figure 1 where the PPBM is divided into column-sets each is of width d. The processing unit in this case (the FDSSM) is responsible for generating and accumulating the partial product digits of one column-set per cycle (starting by the least-significant column-set up to the most significant one.)

The partial product digits generation per cycle is further divided within the FDSSM over n/d identical basic blocks (BB_0, BB_1 … $BB_{(n/d)-1}$) where each basic block is responsible for generating d digits. This is shown in Figure 2 where each basic block is responsible for a row-set of width d of the PPBM. The intersection of the column-set and the row-set defines the partial product digits generated at the relevant cycle by the relevant basic block respectively.

The basic blocks are shown in the upper part of Figure 2 and the lower part shows the detailed design of one basic block. Each basic block has three inputs and two outputs.

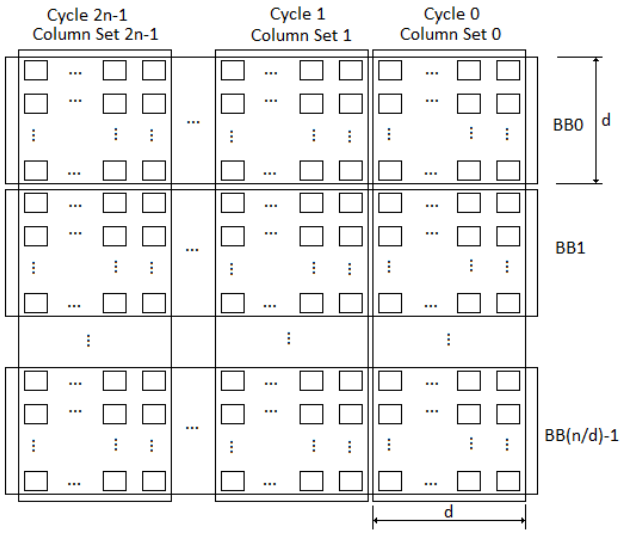


Figure 1: Partial Product Bits per Cycle per Basic Block

The first input of each block is current cycle's digit of the multiplier operand B (indicated by 1 on Figure 2). The control signal of the basic block enables latching this digit within the block such that each block latches only the digit of B respective to the row-set it is responsible for.

The second input is one digit of the multiplicand operand A received from previous block (indicated by 2 on Figure 2), we call it the current digit of A (with respect to the basic block.)

The third input is the previous digit of A (with respect to the basic block) received from next block (indicated by 3 on Figure 2) In summary, digits of A are shifted along the basic blocks such that the current digit of A of a basic block becomes the current digit of A of the next basic block and the previous digit of A of the former basic block at the next cycle.

The need for the previous digit of A can be seen in equation (3) where there is a one bit shift-left per row. This makes each column-set spanning two digits of A: current digit of A and previous digit of A. The feedback of the previous digit of A is needed to complement the shifted versions of the current digit of A. A register of width $(d - 1)$ is used to latch and provide the previous digit of A for the last basic block.

The outputs of the block are the generated partial product digits (indicated by 4 on Figure 2) fed to the accumulation stage and the current digit of A (indicated by 5 on Figure 2) fed to both the next and the previous blocks. This way the unit generates d partial product digits per cycle.

3.2 Partial Product Digits Accumulation

In this section we show how the generated partial product digits are accumulated. As mentioned in section 3.1, the output of each basic block is d partial product digits. Here we use a tree of 4-to-2 compressors to accumulate the generated partial product digits as shown in Figure 3.

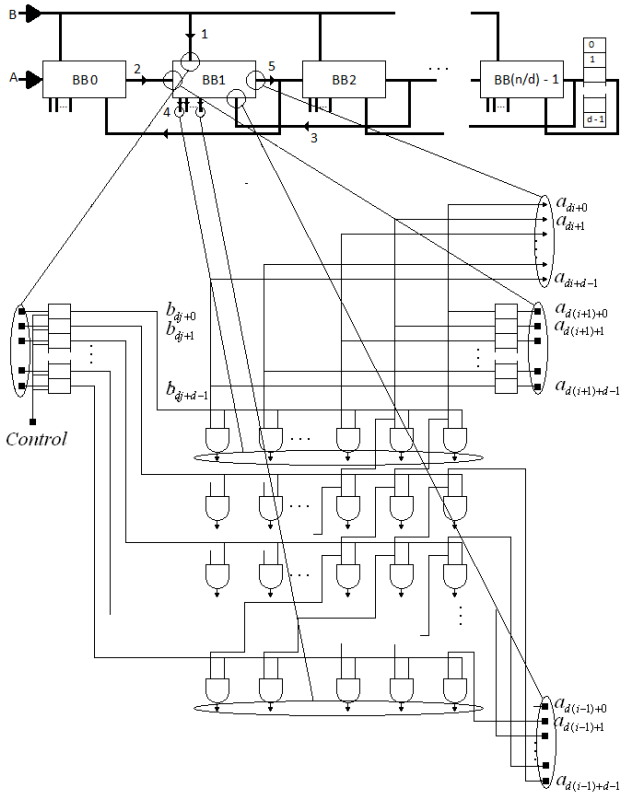


Figure 2: Partial Product Digits Generation Unit

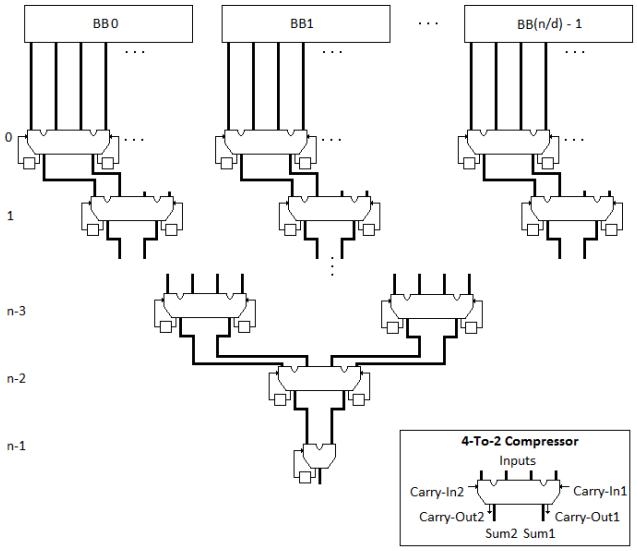


Figure 3: Partial Product Digits Accumulation

Each of the 4-to-2 compressors takes 4 partial product digits and 2 carry-in bits as input and compresses them into 2 partial product digits and 2 carry-out bits. A 4-to2 compressor possesses a delay of $2D_{FA}$, where D_{FA} is one full adder delay. The 2 carry-out bits are latched and routed back as carry-in for the same compressors next cycle. The tree root is a normal

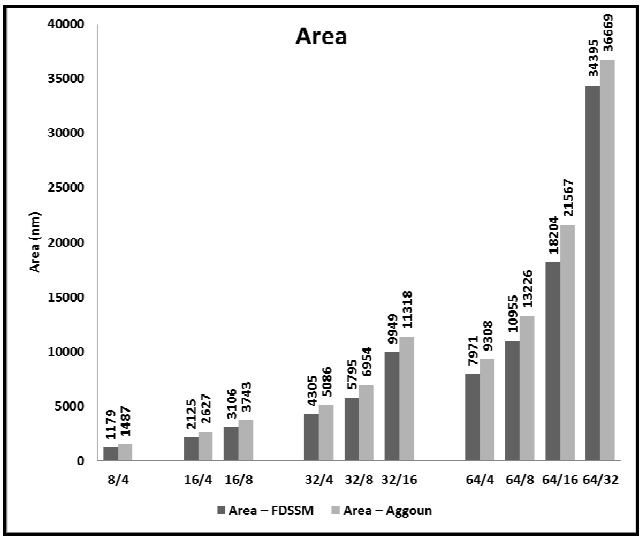


Figure 4: Area Comparison between FDSSM and Agg. [1]

adder that sums two partial product digits into one digit. The tree consists of *(n/2)-1* compressors and *1* normal adder. Each compressor consists of *2d* full adders and the normal adder consists of *d* full adders giving a total of *2d ([n/2]-1) +d = d (n-1)* full adders. The tree latency can be given by: $2D_{FA}$ *([log n] - 1)* $+D_{ADD}$ where D_{FA} is the full adder delay and D_{ADD} is the normal adder latency.

4 Experimental Results

In this section we compare the FDSSM with the multiplier proposed by Aggoun [1]. As mentioned before, their work is the only in literature that proposed a Digit-Serial-Serial multiplier. The work of Almiladi [2] is a slightly modified version for the work of Aggoun [1] that uses a different design methodology. Aggoun [1] proposed two implementations: the first with a basic-block per digit of the operand and the second uses only half the count of basic-blocks and buffers the most significant operands' halves and reroutes them for processing after the least significant halves are processed. The second implementation is the one we compare to since it occupies less chip area and possesses less area-time product.

We implemented the FDSSM and the compared-to Aggoun [1] in VHDL. The implementation was done using Cadence Encounter Digital Implementation RTL Compiler (EDI9.1_ISR4_s273) on CentOS 6 - x368 using NCSU-FreePDK45-1.4 cell library and targeting a clock period of 100ps. The comparison was done with respect to area, time, and area-time product measures. All multiplier instances used equal-width operands (a constraint of Aggoun [1]) Simulation results are shown in Figures 4, 5, 6, 7, 8, and 9. In the experiments, we change operands' width *n* from 8 to 64 bits while changing the digit width *d* from 4 to 32 bits. In the figures, 8/4 means that the operand width *n = 8* bits while the digit width is *d = 4*.

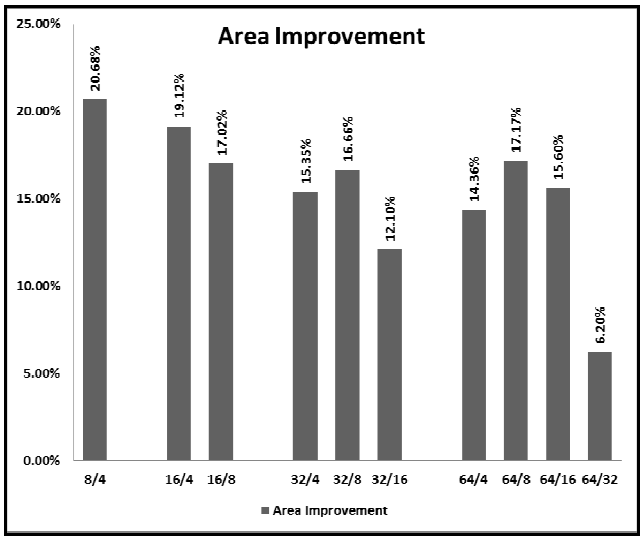


Figure 5: Area Improvement of FDSSM Over Aggoun [1]

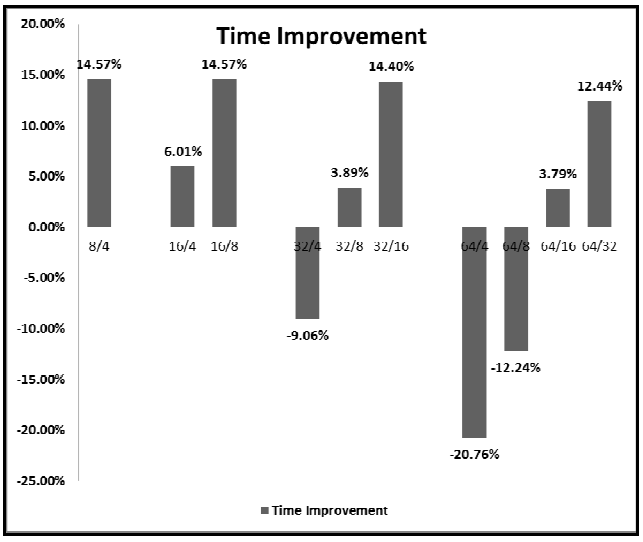


Figure 7: Time Improvement of FDSSM Over Aggoun [1]

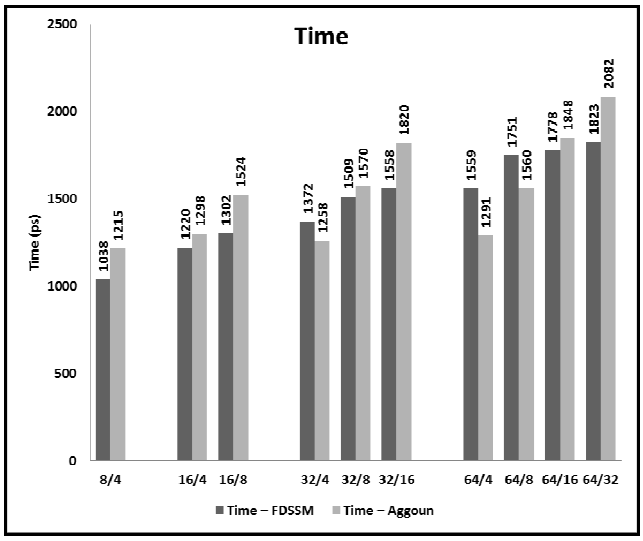


Figure 6: Time Comparison between FDSSM and Agg. [1]

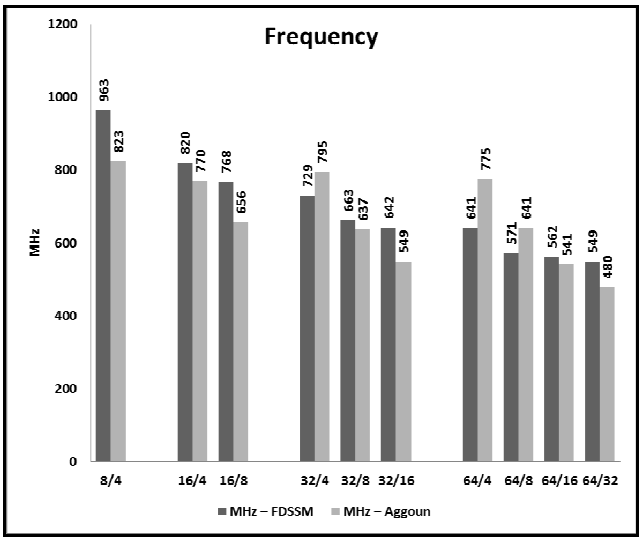


Figure 8: Achievable Running Frequency

Figure 4 shows how the area changes with different operand width and digit width. The area is measured in nanometer.

Figure 5 shows the area improvement percent for the FDSSM over Aggoun [1]). For example, for 32/8, the area is reduced in FDSSM by about 16%. The excess area in Aggoun [1] results from operand buffering done where n buffers are used to buffer the most-significant halves of the operands together with an input MUX of width d. Also the design of Aggoun [1] buffers the previous digit of both operands per unit to complement the shifted partial product digits generated. Both elements are not present in the FDSSM causing the area reduction.

Figure 6 and Figure 7 show the clock cycle time comparison between FDSSM and Aggoun [1]. Theoretically FDSSM outperforms the work in Aggoun [1] in time for n/d ratios less than or equal to 4; that is, operands are split into 4 digits or less. The time required for FDSSM is given by $T_{FDSSM} = ((2\log n)-2+d)\ T_{FA}$ where T_{FA} is the full adder time whereas the time for the design Aggoun [1] is given by $T_{Aggoun\ [1]} = ((2\log d)+2+d)T_{FA}$. Thus, n/d must be ≤ 4 for T_{FDSSM} to be less than or equal to $T_{Aggoun\ [1]}$.

This reflects on the time improvement as seen for cases like 32/4, 64/4, and 64/8 in Figure 7. The figure shows that FDSSM possess time improvement up to 14% for n/d ratios less than or equal to 4. In other cases where this ratio is more than 4, the work of Aggoun [1] outperforms the FDSSM.

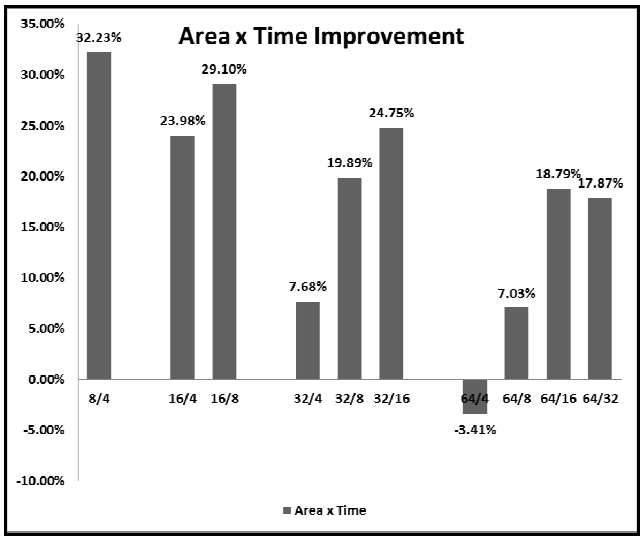


Figure 9: Area-Time Product Improvement

Figure 8 shows the circuit achievable frequency. The FDDSM frequency ranges from about 550 MHz to 960 MHz whereas Aggoun [1] frequency ranges from about 480 MHz to 820 MHz . The frequency graph is the reciprocal of time graph; Aggoun [1] outperforms at 32/4, 64/4, 64/8 whereas FDSSM outperforms at the rest of values.

Figure 9 shows the area-time product comparison between FDSSM and Aggoun [1]. The area improvement compensates for the time lag in the 32/4 and 64/8 cases, and diminishes the time lag in the 64/4 case.

It should be noted that Aggoun [1] imposes a constraint that the digit count must be even (due to using half the digit-count of basic blocks) however FDSSM does not impose this constraint; that is it supports even and odd digit count giving designers more flexibility in area-speed compromises.

5 Properties of the FDSSM

In this section we introduce two properties that the FDSSM possesses. First we introduce the property of dynamic width operand then we show how the FDSSM can be modified to support the two's complement multiplication.

5.1 Dynamic Width Operand

The proposed multiplier latches one operand's digit per cycle and shifts the other operand's digits along the basic blocks. This allows the second operand to be of any digit count (i.e., dynamic.) This enables long operand multiplication by fixed-width operand which is common in cryptography applications. The number of cycles required to complete the multiplication is given by $(n + m) / d$ where n and m are the operand sizes, $m > n$. The number of basic blocks required in this case is equal to the digit count of n. That is the area and time specification of an FDSSM with one dynamic operand

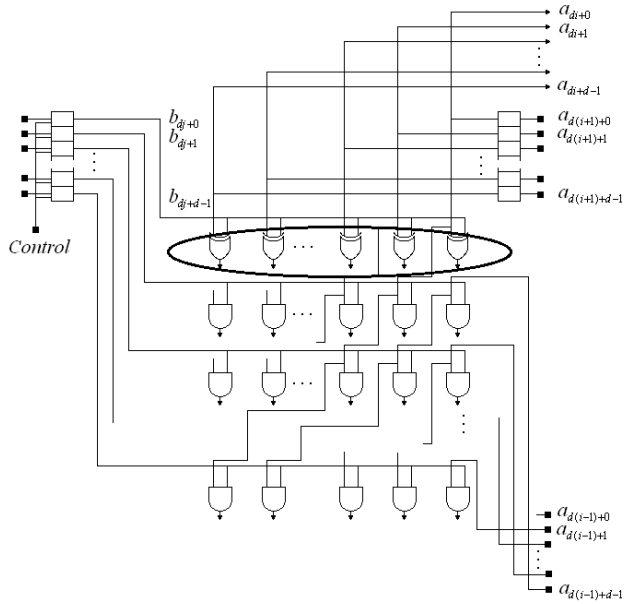


Figure 10: Modifying the Multiplier to Support 2's Complement

matches those of an FDSSM with equal size operands with the operand size equal to the smaller operand.

On the other hand in Aggoun [1], both operands are latched one digit per cycle which constraints the multiplier for fixed, equal-width operands. Accordingly the number of basic blocks required is equal to half the digit count of m; and n is extended to match m.

To support dynamic width operand, another input signal is added to the FDSSM flagging the arrival of the last digit of m. This signal tells the FDSSM that starting next cycle, zeros digits must be injected instead of m digits (no more m digits). The last result digit is produced after n/d cycles of receiving the last digit of m

.

5.2 Two's Complement Support

In this section we show how the FDSSM can be modified to support 2's complement operands. The 2's complement support applies the methodology proposed by Ienne and Viredaz [5]. As mentioned before, one of the operands, the latched operand is considered the multiplier and the second operand, the shifted operand is considered the multiplicand.

To support 2's complement operands, we deal with the multiplicand and the multiplier as follows. For the shifted multiplicand operand, a sign extension is performed. In case of the dynamic-width multiplicand, an extra input is required to indicate the last digit so that sign extension starts next cycle. As for the latched multiplier operand, the basic block of most significant digit is modified replacing the AND gates connected to the most significant bit of the latched digit to XOR gates (highlighted in Figure 10). This has the effect of inverting the bits of multiplicand digits if that bit (i.e., the sign bit) is 1. The first effective carry-in of the adder corresponding to the

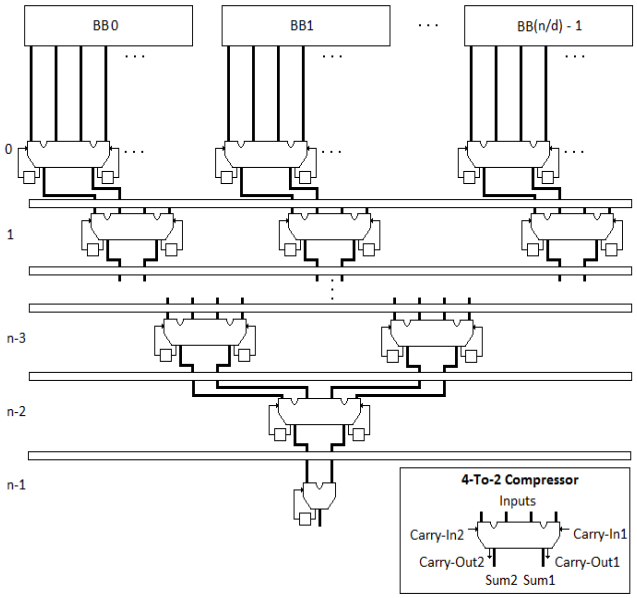


Figure 11: Bit-Level Pipelining Support

modified partial product digit is also reset to 1 rather than 0 if that bit is 1, such that the multiplicand negation is complete (i.e., if the sign bit of the multiplier operand is 1; the multiplicand operand is subtracted rather than added.) Figure 10 shows the modification. This not applicable to Aggoun [1] since the multiplier sign bit is processed at each of the basic blocks compared to FDSSM where it is processed only at the most significant digit's basic block.

5.3 Bit-Level Pipelining Support

A bit-level pipelined multiplier is a multiplier with a pipelined circuit such that the pipeline stage delay is independent of the digit width and the operands width(s). Multipliers with feed-backward lines cannot be bit-level pipelined since the feedback will be out of sync due to pipelining. Also multipliers that use normal digit adders cannot be bit-level pipelined since the pipeline stage delay will be dependent on the adder/digit width.

In the FDSSM, bit-level pipelining can be achieved by latching the compressors' outputs from one level to the next.

Figure **11** shows the modifications required and Figure 12 shows the 4-to-2 compressor circuit [1]. The final adder is replaced by a bit-level pipelined adder as the one described in [7] and shown in Figure 13. This modification makes the pipeline stage delay equal to a D-FF, an AND gate and two FA(s) independent of the digit width.

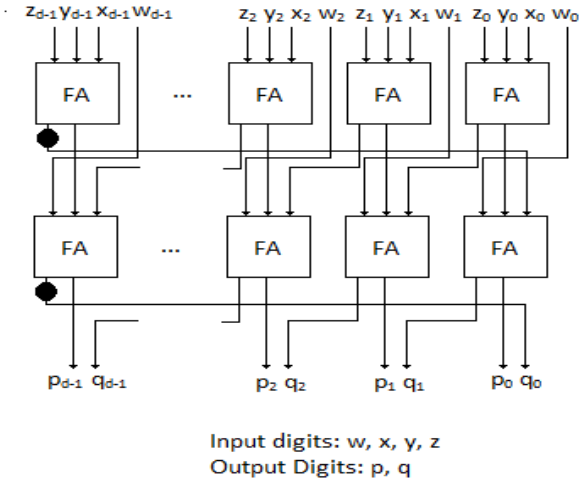


Figure 12: Structure of a 4-To-2 Compressor [1]

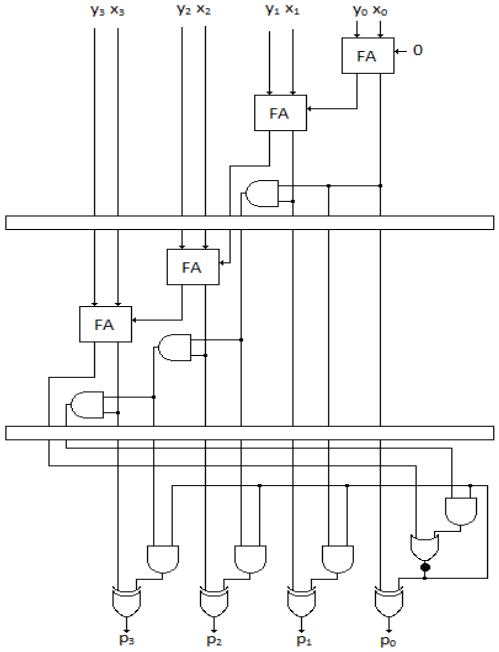


Figure 13: 4 Bit- Bit-Level-Pipelined Adder

6 Concluding Remarks

In this paper a new Digit-Serial-Serial multiplier is proposed that is efficient in both area and area-time product measures. The presented multiplier supports dynamic-width operand while the other-shorter operand is fixed-width. The multiplier was shown to be able to perform 2's complement multiplication and supports bit-level pipelining. Simulation results showed that the proposed multiplier reduces the required area and area-time product significantly.

An extension of this work can be in the design of a Digit-Serial-Serial divider of a similar organization or based on the proposed multiplier. Other direction includes having a full Digit-Serial-Serial ALU as a building block in processors.

References

[1] Aggoun, A.; Farwan, A.F.; Ibrahim, M.K.; Ashur, A., "Radix-2^n serial-serial multipliers," IEE Proceedings - Circuits, Devices and Systems, vol.151, no.6, pp. 503-509, 15 Dec. 2004.

[2] Almiladi, A.; "A Novel Methodology for Designing Radix-2^n Serial-Serial Multipliers," Journal of Computer Science 6 (4): 461-469, 2010.

[3] Dimitrov, V.S.; Jarvinen, K.U.; Adikari, J.; "Area-Efficient Multipliers Based on Multiple-Radix Representations," IEEE Transactions on Computers, vol.60, no.2, pp.189-201, Feb. 2011.

[4] Gnanasekaran, R.; "On a Bit-Serial Input and Bit-Serial Output Multiplier," IEEE Transactions on Computers, vol.C-32, no.9, pp.878-880, Sept. 1983.

[5] Ienne, P.; Viredaz, M.A.; "Bit-serial multipliers and squarers," IEEE Transactions on Computers, vol.43, no.12, pp.1445-1450, Dec 1994.

[6] Lamberti, F.; Andrikos, N.; Antelo, E.; Montuschi, P.; "Reducing the Computation Time in (Short Bit-Width) Two's Complement Multipliers," IEEE Transactions on Computers, vol.60, no.2, pp.148-156, Feb. 2011.

[7] Nibouche, C.; Nibouche, M., "On designing digit multipliers," 9th International Conference on Electronics, Circuits and Systems 2002, vol.3, no., pp. 951- 954 vol.3, 2002.

[8] Nibouche, O.; Bouridane, A.; Nibouche, M.; "New architectures for serial-serial multiplication," The 2001 IEEE International Symposium on Circuits and Systems 2001, ISCAS 2001, vol.2, no., pp.705-708 vol. 2, 6-9 May 2001.

[9] Stelling, P.F.; Martel, C.U.; Oklobdzija, V.G.; Ravi, R.; "Optimal circuits for parallel multipliers," IEEE Transactions on Computers, vol.47, no.3, pp.273-285, Mar 1998.

[10] Wu, C.W. and P.R. Cappello, 1989. "Block multipliers unify bit-level cellular multiplications," Int. J. Comp. Aid. VLSI Des., 1: 113-125.

RB_DSOP: A Rule Based Disjoint Sum of Products Synthesis Method

P. Balasubramanian*
Department of Electronics and Communication Engineering,
S.A. Engg College (aff to Anna Univ),
Chennai 600 077, TN, India
spbalan04@gmail.com

R. Arisaka
School of Computing,
Teesside University,
Middlesbrough TS1 3BA,
United Kingdom
a_ryuta@yahoo.co.uk

H. R. Arabnia
Department of Computer Science,
University of Georgia,
415 Boyd Building,
Athens, Georgia 30602-7404, USA
hra@cs.uga.edu

***Abstract*—A novel disjoint sum of products (DSOP) synthesis method is presented in this paper. It has been found out from analysis that a pair of logical product terms can subscribe to just four possible logic relationships. This important observation underlies the development of the proposed Rule Based DSOP (RB_DSOP) synthesis scheme that applies specific Boolean rules to transform the overlapping products into non-overlapping ones, in order to deduce the minimum DSOP expression from a reduced sum of products (SOP) form. The RB_DSOP method is implemented using Java, incorporating parallel problem solving capability, and is available as a stand-alone tool for teaching and/or research purposes – free access can be provided upon request. The major highlight of this research being that the cost (number of essential products) of the DSOP solution generated for a number of combinational benchmarks using the proposed RB_DSOP method is found to be significantly lower (77%) in comparison with the cost of the DSOP solution derived using a well-known open-access DSOP routine in the existing literature.**

I. INTRODUCTION

A Boolean product represents a conjunction of distinct literals, where a literal specifies a Boolean variable (x) or its complement (x'). A Boolean function, f, is a mapping of type f: $\{0,1\}^n \rightarrow \{0,1,d\}$, where '$d$' denotes a don't care condition. If d does not exist, then the function f is said to be completely specified or two-valued, otherwise it is incompletely specified. Each of the 2^n nodes in the Boolean space corresponds to a canonical product term (minterm). The ON-set, OFF-set and DC-sets of f correspond to those minterms that are mapped to 1, 0 and d respectively. A Boolean equation can be expressed in the sum-of-products (SOP) or disjunctive normal form, where the products are all irredundant. In general, a SOP is said to contain minimum number of essential products by definition and hence it is also referred to as minimum SOP. A Boolean specification is said to be expressed in the disjoint sum of products (DSOP) form, if it is described by a logical sum of product terms that are all disjoint [2], i.e. no two product terms cover a common minterm in their expanded form – in other words, the products are non-overlapping and their logical conjunction results in a null. A DSOP form with the least number of irredundant product terms is known as minimum DSOP.

While SOP minimization can be likened to a set covering problem, DSOP minimization can be likened to the problem of finding a minimum exact disjoint cover which is NP-hard [2]. For example, the number of essential products comprising the SOP expression of an Achilles' heel function [2] is given by $O\left(n/2\right)$, while the number of essential products constituting its DSOP expression is specified by $O\left(2^{n/2}-1\right)$, where 'n' represents the number of distinct primary inputs. DSOPs have been traditionally used in several applications in CAD areas, for example, to calculate the spectra of Boolean functions [3 - 5], or as a starting point for the minimization of Exclusive-OR SOP logic [6] [7], which forms the backbone of synthesis schemes for reversible logic circuits [8] [9] that assumes significance in the realm of quantum computing. A number of DSOP methods have been proposed by researchers over the past few decades [10] – [14] by relying on heuristics, considering utilization of reduced ordered binary decision diagrams (ROBDDs) or adopting evolutionary programming techniques. Nevertheless, ROBDDs being inherently mutually exclusive may suffer from huge memory space requirements for higher order functionality and therefore heuristics might be preferable. Of these, the Espresso_DSOP routine [1] is widely referred and it is primarily an open-access tool. In general, DSOP solutions generated by Espresso are far from optimum; this is more so the case for functions with several concurrent outputs as the synthesis scheme resorts to group minimization of all the function outputs. An alternative approach would be to consider deriving DSOP solutions for the function outputs individually on the basis of their SOP forms, which is the strategy adopted in case of our RB_DSOP method.

The remaining part of this paper is organized as follows. Section 2 highlights the various pair-wise logical product scenarios and discusses how select Boolean axioms are applied to transform the non-disjoint products into disjoint ones. Section 3 describes the RB_DSOP synthesis scheme from a high-level perspective, and Section 4 tabulates the SOP and DSOP costs pertaining to a large set of combinational logic benchmarks based on Espresso_DSOP and RB_DSOP methods. Lastly, the conclusions are arrived at in Section 5.

* This research work was performed when the author was affiliated with the Department of Electronics and Communication Engineering, Vel Tech Technical University, Avadi, Chennai 600 062, TN, India.

II. Pair-Wise Logical Products – Classification

In this section, we classify logical product pairs into four categories and describe ways of converting non-disjoint products into disjoint ones – disjoint products are non-overlapping while non-disjoint product terms tend to overlap. It should be noted that the examples used for describing the subsequent logic transformations are mainly representative of general function scenarios. A pair of products are said to be overlapping if they cover a common canonical product (cube) when expanded. A canonical product term is also called as *minterm*, which is a unique conjunction of all input variables.

A. Pair-Wise Product Scenario 1: $F(a,b,c) = ab + c$

A Boolean function $F(a,b,c)$, dependent on three binary variables, is composed of two product terms, say $P_1 = ab$ and $P_2 = c$. The *support* (*set*) of a product term enumerates the variables constituting it. Here the support set of product terms P_1 and P_2 contain distinct variables, i.e. $s(P_1) = \{a,b\}$ and $s(P_2) = \{c\}$, implying $s(P_1) \cap s(P_2) = \phi$. Products P_1 and P_2 are said to be overlapping as they cover the common minterm abc in their expanded form, and hence they are non-disjoint.

Citing this scenario, we describe how to convert two overlapping product terms into non-overlapping (disjoint) ones, where the products consist of unique variable conjunctions with one of the product terms being a singleton. Product term P_2 is said to be singleton as $|s(P_2)| = 1$. In this case, the converse of the *absorption axiom* $(x + x'y = x + y)$ is used to transform the product term P_1 as thus: $P_1 = abc'$. Now the two product terms become disjoint and they would not overlap since they no more cover a similar minterm in their canonical forms. Therefore the transformed Boolean function F^*, which is logically equivalent to the original function F is expressed as, $F^*(a,b,c) = abc' + c$. Here, F and F^* exhibit combinational equivalence and they represent the respective SOP and DSOP function formats.

B. Pair-Wise Product Scenario 2: $F(a,b,c) = ab + ac$

A Boolean function $F(a,b,c)$, dependent on three binary variables, is composed of two product terms, say $P_1 = ab$ and $P_2 = ac$. Here the support of product terms P_1 and P_2 contain the common variable a. In terms of set notations, $s(P_1) = \{a,b\}$ and $s(P_2) = \{a,c\}$ and $s(P_1) \cap s(P_2) \neq \phi$. Again, product terms P_1 and P_2 are found overlapping as they cover the common minterm abc when expanded, and so they are not disjoint.

In cognizance of this scenario, we describe how to convert two non-disjoint product terms into a disjoint product pair, where the products share one or more common variables, with neither of the product terms being a singleton. Use the *distributive axiom* of Boolean algebra viz. $ab + ac = a(b + c)$ to extract the kernel $(b + c)$. The converse of the *absorption axiom* of Boolean algebra is applied to transform the kernel comprising overlapping product terms (bc is shared within the kernel) into non-overlapping products as mentioned above. Thus the kernel $(b + c)$ is transformed into $(b + b'c)$. Using the *distributive property* of Boolean algebra, $x(y + z) = xy + xz$, re-enumerate the product terms in the disjunctive normal form. The transformed logic function is, $F^*(a,b,c) = ab + ab'c$. Here again, F and F^* feature combinational equivalence, and they signify the respective SOP and DSOP function formats.

C. Pair-Wise Product Scenario 3: $F(a,b,c,d) = ab + cd$

A Boolean function $F(a,b,c,d)$, dependent on four binary variables consists of two product terms, given by $P_1 = ab$ and $P_2 = cd$. The support set of products P_1 and P_2 do not contain any common variable. As per set notations, $s(P_1) = \{a,b\}$ and $s(P_2) = \{c,d\}$ and $s(P_1) \cap s(P_2) = \phi$. Product terms P_1 and P_2 are overlapping since they cover the common minterm $abcd$ in their expanded form, and hence they are non-disjoint.

This illustration depicts another different scenario where the product terms consist of unique variable conjunctions with neither of the products being a singleton, i.e. $|s(P_1)| \neq 1$ and $|s(P_2)| \neq 1$. In this case, the *identity axiom* $(x + x' = 1)$ of Boolean algebra is applied to any of the pair of products considered; let us say to product term P_2 as $(a + a')cd$. Following this, the product terms are expanded by applying the distributive axiom as $(acd + a'cd)$. Now the transformed logic function is given by the logical sum of three product terms P_1, P_2 and P_3 respectively as: $F^*(a,b,c,d) = ab + acd + a'cd$. It can be seen that P_1 and P_2 are mutually disjoint with P_3, and P_2 and P_3 are also disjoint. With P_1 and P_2 being non-disjoint, using the distributive axiom the kernel $(b + cd)$ is extracted as described in sub-section 2(B), which is then transformed into $(b + b'cd)$ as discussed in sub-section 2(A). Distributive law is again applied to enumerate the product terms comprising the transformed function, which is given by $F^*(a,b,c,d) = ab + ab'cd + a'cd$. Again, we find that $F(a,b,c,d)$ and $F^*(a,b,c,d)$ are combinatorially (logically) equivalent, although being algebraically different.

D. Pair-Wise Product Scenario 4: Covering Term and Covered Term, Redundant Products

On account of the logic optimizations detailed above to translate a function expressed in SOP form into a DSOP form, where all the product terms are mutually exclusive, the possibility of covering term and covered terms might result. Also, redundant products may crop up. In case of the former, *absorption law* of Boolean algebra is used to eliminate the covered products, i.e. $x + xy = x$. In case of the latter, logic duplication is avoided through the *idempotency law*: $x + x = x$.

III. RB_DSOP Synthesis Method

The proposed RB_DSOP synthesis heuristic is explained through the following steps. A simple lexicographical variable ordering is followed for transformations or optimizations involving the product terms of a Boolean function.

- Step 1: Obtain the SOP form of a logic function
- Step 2: Compare each product term with every other product term in the SOP expression to check whether they are mutually disjoint. If so, go to Step 8, else proceed with Step 3
- Step 3: Enumerate all the overlapping pairs of products which do not feature any common variables but with one of the product terms being a singleton (as highlighted in *Scenario 1*). Transform such product term pairs into mutually exclusive ones as discussed in Section 2(A). If *Scenario 1* does not arise, proceed with Step 4

- Step 4: Enumerate all the overlapping pairs of product terms, where the product terms feature variable sharing but with neither of the products being a singleton (as outlined by *Scenario 2*). Transform such logical products into mutually exclusive pairs as discussed in Section 2(B). If *Scenario 2* does not arise, proceed with Step 5
- Step 5: Enumerate all the overlapping product pairs, which comprise distinct variable supports with neither of the product terms being a singleton (as portrayed by *Scenario 3*). Convert such product pairs into disjoint ones based on the procedure described in Section 2(C)
- Step 6: If any logical product is found to cover any other product term in the transformed function expression and/or if logic duplication occurs (as depicted by *Scenario 4*), eliminate the covered term and discard logic repetitions as mentioned in Section 2(D)
- Step 7: Return back to Step 2 to ensure the SOP expression is transformed into a DSOP format
- Step 8: Terminate the synthesis routine as the DSOP form of the requisite logic specification has been obtained

In general, a logic function would have several concurrent inputs and outputs. The RB_DSOP synthesis procedure is applied simultaneously to all the SOP expression(s) of the output(s) of a logic function – independently and in parallel. The reduced SOP form of a Boolean function is obtained by means of a standard open-access logic minimizer, Espresso [1]. The logical correctness of the DSOP solution derived is guaranteed by the Boolean rules applied, which are indeed well-established and proven properties, while the functional correctness of the DSOP solution is ensured by comparing each product with every other product term forming the cover of a function's primary output. However, such a comparison is performed as part of data processing. The combinational equivalence of a SOP form and its DSOP form is confirmed through the *Dverify* option of Espresso.

The *cost* of the DSOP solution derived for a logic function, starting from its SOP specification, is specified by the count of unique product terms, some/all of which may eventually be shared between the various function outputs. Depending upon the initial logic description, several iterations of some/all of the synthesis steps may be required to deduce the minimum DSOP form. The RB_DSOP synthesis package basically reads an input file described in the conventional programmable logic array (.pla) format and produces an output file in a custom-defined format which provides the following information: DSOP equations obtained corresponding to the individual primary outputs, cost factor of the respective primary outputs, cost of the entire function taking into account logic sharing if any, and description of the outputs in .pla style.

IV. Experimental Results

The synthesis procedure of the proposed RB_DSOP heuristic, elucidated in Section 3, has been automated using Java and is configured to be a stand-alone executable (tool). Parallelism is made implicit in the RB_DSOP software package in that the tool can simultaneously solve multiple problems with ease – this attribute of parallel problem solving by use of multiple threads does not appear to be present in any DSOP package known to the authors. To highlight the efficacy of the RB_DSOP method over the well-known and widely referred Espresso_DSOP routine, we will first look at two function examples. Firstly, we consider a small combinational benchmark specification, *newill*, which comprises 8 primary inputs and a single primary output. The SOP and DSOP expressions corresponding to this benchmark function are specified by equations (1) – (3).

$$newill_{SOP} = bc'e'f'gh + bdef'gh + bcd'f'gh + bd'efg'h + bcefg'h + bce'f'g'h + d'e'f'g'h + a \quad (1)$$

$$\begin{aligned} newill_{Espresso\_DSOP} = {} & bc'e'f'gh + bdef'gh + bcd'f'gh + bd'efg'h \\ & + bcdefg'h + bce'f'g'h + b'd'e'f'g'h \\ & + bc'd'e'f'g'h + ab'd + ab'd'e + abc'def \\ & + abc'e'f + abcde'f + ab'd'e'f + abcd'e'f \\ & + abc'd'ef' + abcdefg + abcd'efg \\ & + abc'd'efg + abcde'f'g + ab'd'e'f'g \\ & + abcdef'g' + abc'def'g' + abcd'ef'g' \\ & + abc'de'f'g' + abcd'f'gh' + abcdef'gh' \\ & + abc'def'gh' + abc'e'f'gh' + abcdefg'h' \\ & + abcd'efg'h' + abc'd'efg'h' \\ & + abcde'f'g'h' + ab'd'e'f'g'h' \\ & + abcd'e'f'g'h' + abc'd'e'f'g'h' \end{aligned} \quad (2)$$

$$\begin{aligned} newill_{RB\_DSOP} = {} & a'bc'e'f'gh + a'bdef'gh + a'bcd'f'gh \\ & + a'bc'd'efg'h + a'bcefg'h + a'bcde'f'g'h \\ & + a'd'e'f'g'h + a \end{aligned} \quad (3)$$

The minimized SOP form consists of 8 products, while the RB_DSOP and Espresso_DSOP methods yield 8 and 36 disjoint product terms respectively. The *cost* (quantified as number of irredundant products) of a DSOP expression may be equal to (best case solution) or greater than (typical case solution) the cost of a corresponding SOP form. From (1) – (3), it is clear that the RB_DSOP method has enabled the best possible (exact minimum) solution for the combinational benchmark, *newill*. In comparison with the Espresso_DSOP scheme, the proposed method has effected an excellent cost reduction of 77.8% for this case study.

Let us now consider a multiple input, multiple output combinational benchmark functionality, *clpl*, which consists of 11 primary inputs (*a,b,c,d,e,f,g,h,i,j,k*) and 5 primary outputs (*Y1,Y2,Y3,Y4,Y5*). Its minimized SOP expressions are,

$$Y1_{SOP} = aefg + cfg + dg + b \quad (4)$$

$$Y2_{SOP} = aef + cf + d \quad (5)$$

$$Y3_{SOP} = ae + c \quad (6)$$

$$Y4_{SOP} = aefgij + cfgij + dgij + bij + hj + k \quad (7)$$

$$Y5_{SOP} = aefgi + cfgi + dgi + bi + h \quad (8)$$

The corresponding DSOP forms deduced on the basis of the proposed RB_DSOP method are given as,

$$Y1_{\text{RB\_DSOP}} = ab'c'd'efg + b'cd'fg + b'dg + b \quad (9)$$

$$Y2_{\text{RB\_DSOP}} = ac'd'ef + cd'f + d \quad (10)$$

$$Y3_{\text{RB\_DSOP}} = ac'e + c \quad (11)$$

$$Y4_{\text{RB\_DSOP}} = ab'c'd'efgh'ijk' + b'cd'fgh'ijk' + b'dgh'ijk' + bh'ijk' + hjk' + k \quad (12)$$

$$Y5_{\text{RB\_DSOP}} = ab'c'd'efgh'i + b'cd'fgh'i + b'dgh'i + bh'i + h \quad (13)$$

From equations (4) – (13), it is evident that the cost of SOP and DSOP forms (derived using the RB_DSOP method) for the benchmark function, *clpl* are similar – 20 products. Thus the proposed method has once again facilitated the best solution for a logic function. The cost of the DSOP format obtained using the Espresso_DSOP routine is found to be 231 (product terms), and they are too many to detail here. Hence the interested reader is encouraged to check out the list of products by running the Espresso_DSOP routine on the benchmark, *clpl*. By comparing the cost metrics of DSOP expressions resulting from Espresso_DSOP and RB_DSOP synthesis methods, it can be observed that the latter achieves massive cost reduction to the tune of 91.3% over the former.

The preceding examples drive home the point that the proposed RB_DSOP synthesis scheme has the potential to yield exact DSOP solutions for single/multi-output functions. Experimentation was carried out involving a large number of combinational benchmarks from the MCNC benchmark suite and the results obtained, which are shown in Table 1, duly demonstrate the efficiency of our tool. The cost of respective SOP and DSOP forms of various combinational benchmarks based on Espresso_DSOP and RB_DSOP synthesis heuristics are given in Table 1. Here the cost metric purely represents the number of unique product terms, which implies that some/all of the products might be found, shared between the various primary outputs of a multi-output logic function. For example, in case of benchmarks *mainpla* and *xparc*, the number of product terms constituting the DSOP equation derived on the basis of the RB_DSOP synthesis method is found to be 6853 and 2757 respectively, with no consideration of logic sharing between their individual primary outputs. However, taking cognizance of common logic that is shared between the various primary outputs, the above costs sharply decrease to 283 and 287 product terms respectively.

V. CONCLUSIONS

DSOPs have been traditionally used in several applications in CAD areas, for example to calculate the spectra of Boolean functions [3 – 5], or as a starting point for the minimization of Exclusive-OR SOP logic [6] [7], which forms the backbone of synthesis schemes for reversible logic circuits [8] [9] and assumes significance in the field of quantum computing. While many DSOP synthesis strategies exist in the literature [10 – 14] they are hardly open-access; but the DSOP routine of Espresso [1] is an exception in this regard. This in fact has motivated the authors to collaborate and develop an efficient open-access DSOP synthesis tool, incorporating parallel problem solving capability for easy use/replication by the scientific community. In this context, a novel DSOP synthesis heuristic (RB_DSOP) based on well-founded Boolean rules was propounded in this work. The RB_DSOP synthesis scheme has been implemented using Java on a Windows platform and is available for use as a stand-alone executable (software package) for teaching and/or research purposes – unrestricted access can be provided upon request.

To demonstrate the qualitative efficiency of our package vis-à-vis the open-access Espresso_DSOP synthesis routine, experimentation was carried out on numerous combinational benchmarks. From the results shown in Table 1, it can be inferred that Espresso_DSOP solutions are 9.5× bigger than the reduced SOP solutions, while RB_DSOP solutions are just 2.2× bigger than their SOP counterparts. Hence the latter facilitates a whopping cost reduction of the order of 77% in comparison with the former. However it is worth noting here that heuristics, in general, do not always guarantee exact minima and devising methods to obtain the same is construed to be practically infeasible.

REFERENCES

[1] R.K. Brayton, G.D. Hachtel, C.T. McMullen, A.L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, MA, 1984.

[2] T. Sasao (Ed.), *Switching Theory for Logic Synthesis*, Kluwer Academic Publishers, Dordrect, The Netherlands, 1999.

[3] B.J. Falkowski, "Calculation of Rademacher-Walsh spectral coefficients for systems of completely and incompletely specified Boolean functions," *Proc. IEEE ISCAS,* vol. 3, pp. 1698-1701, 1993.

[4] B.J. Falkowski, C.-H. Chang, "Paired Haar spectra computation through operations on disjoint cubes," *IEE Proc. Circuits, Devices and Systems*, vol. 146, no. 3, pp. 117-123, August 1999.

[5] M.A. Thornton, R. Drechsler, D.M. Miller, *Spectral Techniques in VLSI CAD*, Kluwer Academic Publishers, Boston, MA, 2001.

[6] T. Sasao, "EXMIN2: a simplification algorithm for exclusive-OR-sum-of-products expressions for multiple-valued-input two-valued-output functions," *IEEE Trans. on CAD*, vol. 12, no. 5, pp. 621-632, 1993.

[7] A. Mishchenko, M. Perkowski, "Fast heuristic minimization of exclusive-sums-of-products," *Proc. Intl. Workshop on Applications of Reed-Muller Expansion in Circuit Design*, pp. 242-250, 2001.

[8] D. Maslov, "Efficient reversible and quantum implementations of symmetric Boolean functions," *IEE Proc. Circuits, Devices and Systems*, vol. 153, no. 5, pp. 467-472, October 2006.

[9] P. Gupta, A. Agrawal, N.K. Jha, "An algorithm for synthesis of reversible logic circuits," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 25, no. 11, pp. 2317-2330, November 2006.

[10] B.J. Falkowski, I. Schafer, C.-H. Chang, "An effective computer algorithm for the calculation of disjoint cube representation of Boolean functions," *Proc. 36th IEEE MWSCAS,* pp. 1308-1311, 1993.

[11] L. Shivakumaraiah, M.A. Thornton, "Computation of disjoint cube representations using a maximal binate variable heuristic," *Proc. 34th IEEE Southeastern Symposium on System Theory*, pp. 417-421, 2002.

[12] G. Fey, R. Drechsler, "Utilizing BDDs for disjoint SOP minimization," *Proc. 45th IEEE MWSCAS,* vol. 2, pp. II-306- II-309, 2002.

[13] N. Drechsler, M. Hilgemeier, G. Fey, R. Drechsler, "Disjoint sum of product minimization by evolutionary algorithms," in *EvoWorkshops 2004*, G.R. Raidl et al. (Eds.), *LNCS*, vol. 3005, pp. 198-207, 2004.

[14] A. Bernasconi, V. Ciriani, F. Luccio, L. Pagli, "A new heuristic for DSOP minimization," *Proc. 8th Intl. Workshop on Boolean Problems*, pp. 169-174, 2008.

TABLE I. ENUMERATING THE COST OF SOP AND DSOP FORMS OF VARIOUS COMBINATIONAL BENCHMARKS

Combinational benchmark	# Primary inputs	# Primary outputs	SOP cost (# products)	Espresso_DSOP cost (# products)	RB_DSOP cost (# products)
9sym	9	1	86	209	179
alu4	14	8	575	3549	1206
amd	14	24	66	93	73
apex1	45	45	206	12667	352
apex3	54	50	280	3418	402
apex4	9	19	435	536	507
b2	16	17	106	331	123
b3	32	20	211	8832	330
b10	15	11	100	983	115
b12	15	9	42	654	62
bc0	26	11	178	1894	222
bcd	26	38	117	134	126
chkn	29	7	140	2601	202
clip	9	5	120	359	167
cordic	23	2	914	22228	6687
cps	24	109	163	895	227
dist	8	5	123	236	135
duke2	22	29	86	181	122
ex4	128	28	279	2090	594
ex5	8	63	72	184	142
ex1010	10	10	284	1331	976
exep	30	63	110	294	128
gary	15	11	107	810	120
ibm	48	17	173	2259	353
in3	35	29	74	864	107
in4	32	20	212	8207	326
intb	15	7	631	3533	1055
jbp	36	57	122	862	132
mainpla	27	54	171	1941	283
max512	9	6	145	386	192
max1024	10	6	272	776	362
misex3	14	14	677	3789	1557
misg	56	23	69	7842	106
mish	94	43	82	147	90
mlp4	8	8	127	206	155
mp2d	14	14	31	625	59
newapla	12	10	17	86	27
newcond	11	2	31	60	37
newcpla1	9	16	38	145	52
opa	17	69	79	141	106
pdc	16	40	135	994	485
root	8	5	57	154	60
ryy6	16	1	112	272	112
sao2	10	4	58	199	120
seq	41	35	336	1040	378
soar	83	94	353	1859	478
spla	16	46	253	1119	360
sym10	10	1	210	367	348
t1	21	23	102	662	154
t481	16	1	481	2139	1302
table3	14	14	175	249	175
table5	17	15	158	336	162
test3	10	35	541	1778	1637
ti	47	72	213	2067	351
vg2	25	8	110	863	164
x1dn	27	6	110	706	160
x6dn	39	5	82	1144	126
x7dn	66	15	538	1697	1228
xparc	41	73	254	942	287
z9sym	9	1	85	190	171
Average cost (# products)			**201.9**	**1919.3**	**440.9**

Analysis of Notebook Computer Chassis Design for Hard Disk Drive and Speaker Mounting

J . Q. Mou, Fukun Lai, I. B. L. See, and W. Z. Lin

Data Storage Institute, 5 Engineering Drive 1, Singapore 117608

Abstract - *Chassis design is a critical issue for the reliability and robustness of a notebook personal computer (PC) and its key components, especially hard disk drive (HDD). In particular, oscillating force induced by the build-in speakers in the notebook PC, could be transmitted to the HDD via chassis causing position error signal (PES) of HDD. In this paper, the chassis design, including mounting of the speaker and HDD, is studied. Firstly, a finite-element-method (FEM) model of a commercial notebook PC is developed, consisting of the chassis, HDD, build-in speakers and other key components. Characteristics of HDD vibration induced by the speaker in the notebook PC are examined with the FEM model. Then a simplified theoretical model consisting of lumped mass, spring, and damping elements is developed. This model is implemented to analyze the vibration transmissions in the notebook PC. The effects of the chassis design, speaker and HDD mounting on the vibration transmission from speaker to HDD via chassis in the notebook PC are investigated. Finally, a guideline for appropriate notebook PC chassis design and mounting of speaker and HDD is proposed.*

Keywords: Notebook computer, Chassis design, Hard disk drives (HDDs), Speaker, Vibration, FEM.

1. Introduction

As one type of the most popular personal computers (PC), notebook PC gives the flexibility to work and play anywhere. Therefore, the chassis and components of the notebook PCs should be designed to be robust enough to ensure the diversification of the operating environment. The hard disk drives (HDDs), as the large capacity storage devices, play an important role in the notebook computers. However, HDDs are sensitive to external vibration, which may introduce the position error signal (PES) of magnetic head in data reading and writing. The vibration sources of a notebook PC, as shown in Figure 1, include build-in speakers, cooling fans, CD-ROM drive, touching force at keypad etc. Therefore appropriate chassis design is critical for reducing the vibration transmission from the vibration sources to HDD in notebook PC.

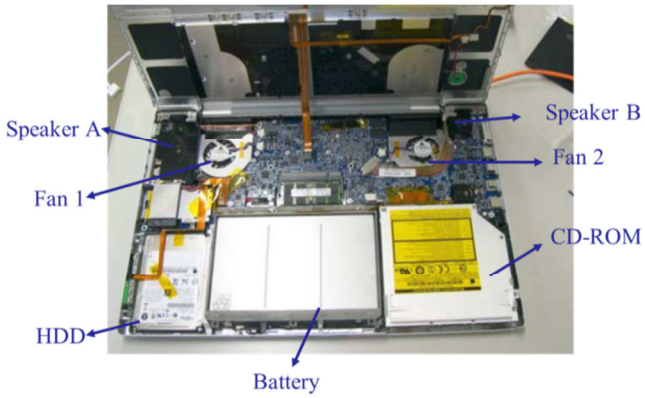


Figure 1. Interior view of a notebook PC

A number of papers have been published on vibrations of HDDs and PCs. For example, G. Ferretti *et al.* analyzed the vibration of HDDs with a dynamic model and experimental measurements [1]. S. Lim *et al.* analyzed dynamic characteristics and shock response of the HDDs and optimized the design of rubber mounts in notebook [2]. Y. Matsuda *et al.* designed, fabricated and evaluated a flexible support mechanism for HDDs to intercept external vibration [3]. T. Semba *et al.* studied an adaptive cancellation method for vibrations of soft mounting HDDs in notebook PC [4]. However, there is very limited published research work on the vibration transmission induced by the build-in speakers, which is one of the major vibration sources in notebook PC. Among them, Y. Y. Hu *et al.* systematically investigated the built-in speakers induced structural-acoustic vibration of HDD in notebook PC by the FEM analysis and experimental studies, and concluded that the acoustically transmitted vibration of speaker can be ignored compared to the structurally transmitted vibration, and softer supporters for HDD mounting is a better choice for reduction of HDD vibration [5].

In this paper, the structural vibration transmission from the build-in speaker to HDD in notebook PC is studied. Firstly, a FEM model of a commercial notebook PC including the chassis mounted with HDD, build-in speakers and other key components is built up with ANSYS software. The characteristics of HDD vibrations induced by the speakers in the notebook PC are examined with the FEM model. Based on the characteristics of HDD vibrations, a simplified theoretical vibration model of the notebook is developed, which is described by a set of ordinary differential equations (ODEs) governing the motion of the speaker, chassis and HDD. The ODEs are solved for analysis of the vibration transmission from speaker to HDD by varying the stiffness and damping of the components. The effects of the chassis design, speaker and HDD mounting on the vibration transmission from speaker to HDD via chassis in the notebook PC are investigated. Finally, a guideline for appropriate notebook PC chassis design and mounting of speaker and HDD is proposed.

2. HDD vibrations induced by speakers in Notebook PC

A FEM model of a commercial notebook PC is developed with ANSYS to examine the characteristics of HDD vibration induced by build-in speakers. The notebook PC is mounted with one HDD and two speakers denoted as speaker A and B, as shown in Figure 1. The FEM model of the notebook, as displayed in Figure 2, consists of a chassis mounted with one HDD, two speakers, and other key components including the LCD monitor, keypad, battery and battery fixture, PCBs, radiator block, CD/DVD, two fans, HDD supporters with damper and bracket connectors etc. There are totally 200,391 nodes and 531,811 elements used in the FEM model. The FEM model has been validated in [5] by comparing simulation results with experimental results.

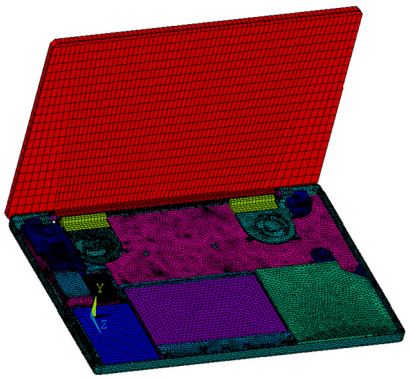


Figure 2. FE model of internal components of the PC

The mode superposition method is used for analysis of the vibrations induced by the speaker. The experimental out of plane velocity frequency response at the speaker rim [5], subject to a sweep sinusoidal current $i=i_0 sin(2\pi ft)$, is used as the excitation input in the FEM vibration analysis.

Figure 3 shows the out of plane vibration response of the speaker A, chassis, and HDD from the FEM analysis with 300 modes subject to the oscillating excitation exerted on the speaker A.

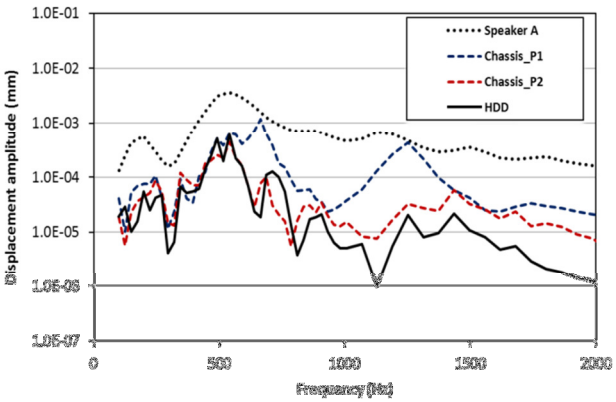


Figure 3. Displacement responses of speaker, chassis and HDD excited by speaker A

The responses of the chassis are represented by two points, one denoted as P1 from the node under speaker A, and the other denoted as P2 from the node under the HDD. As observed in Figure 3, the first dominant response is at the frequency of speaker excitation, and the amplitude of the responses of chassis_P1, chassis_P2 and HDD are decreased gradually due to the vibration energy dissipation along the transmission routes.

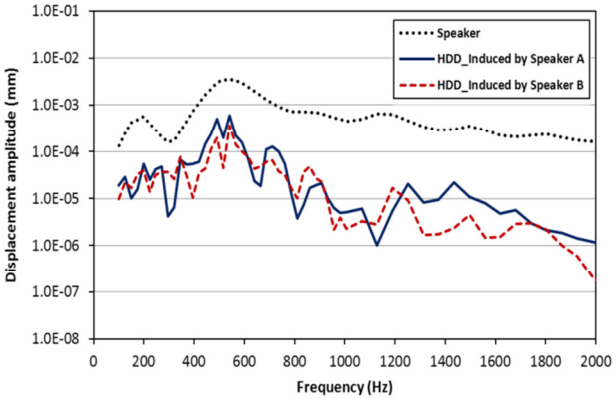


Figure 4. HDD vibration excited by speaker A and B respectively

Figure 4 shows the HDD vibration induced by speaker A and B respectively. The speaker B is positioned far away from HDD and its mounting stiffness is calculated 21.8% smaller than that of speaker A. As shown in Figure 4, the vibration displacement response of HDD induced by speaker A is larger than that by speaker B, especially at frequencies higher than 1000 Hz.

3. Analysis of vibration transmission in Notebook PC

Apart from the numerical FEM model, the theoretical model is intended to give a fundamental understanding of the vibration transmission in notebook PC. In this section, the simplified theoretical vibration model for fundamental analysis of the vibration transmission route from speaker to HDD via chassis in the notebook PC is developed.

3.1 Analytical model development

According to the characteristics of HDD vibration as discussed in FEM analysis, a simplified schematic of vibration transmitted from the build-in speaker to the HDD via chassis in the notebook PC can be extracted, as shown in Figure 5.

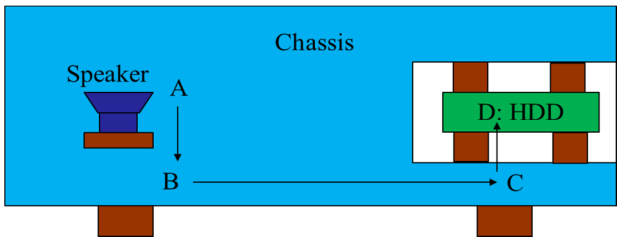


Figure 5. Notebook schematics with speaker and HDD

From Figure 5, it is shown that the vibration is transmitted from speaker to chassis (route AB), and then from one side of the chassis to the other side (route BC), finally from chassis to HDD (route CD). The analytical model is therefore developed on the basis of the three transmission routes, which consists of four degree of freedoms, including speaker, HDD, and chassis, with different mounting methods between them. As shown in Figure 6, the HDD and speaker are modeled as two masses mounted on the chassis through spring and damping elements. The chassis is separated into two parts, i.e., chassis_p1 and chassis_p2 which support the speaker and HDD respectively. Both parts of the chassis are supported by the ground. It is noted that only one direction vibration is considered in the present model. In addition, it is assumed that an external force is exerted on the speaker, when it is playing music or other sounds.

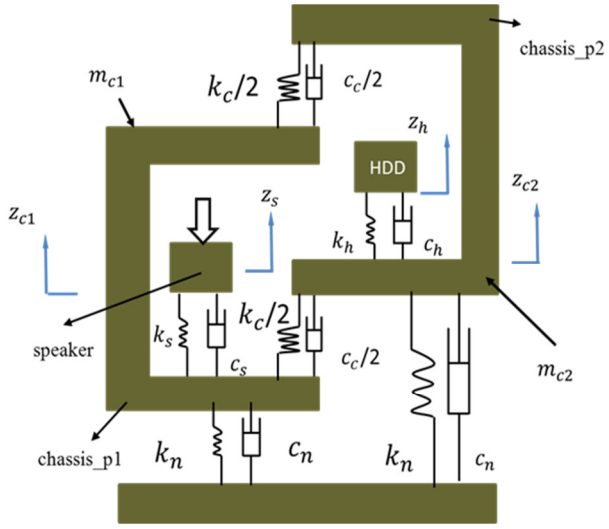


Figure 6. Vibration model of notebook PC

Based on the mass-spring-damper model shown in Figure 6, the ODEs for the vibration motion of speaker, chassis and HDD are developed.

$$m_s\ddot{z}_s + c_s(\dot{z}_s - \dot{z}_{c1}) + k_s(z_s - z_{c1}) = f_s \quad (1)$$

$$m_{c1}\ddot{z}_{c1} + c_c(\dot{z}_{c1} - \dot{z}_{c2}) + c_s(\dot{z}_{c1} - \dot{z}_s) + c_n(\dot{z}_{c1} - \dot{z}_n) + k_s(z_{c1} - z_s) + k_n(z_{c1} - z_n) + k_c(z_{c1} - z_{c2}) = 0 \quad (2)$$

$$m_{c2}\ddot{z}_{c2} + c_c(\dot{z}_{c2} - \dot{z}_{c1}) + c_h(\dot{z}_{c2} - \dot{z}_h) + c_n(\dot{z}_{c2} - \dot{z}_n) + k_h(z_{c1} - z_h) + k_n(z_{c2} - z_n) + k_c(z_{c2} - z_{c1}) = 0 \quad (3)$$

$$m_h\ddot{z}_h + c_h(\dot{z}_h - \dot{z}_{c2}) + k_h(z_h - z_{c2}) = 0 \quad (4)$$

where m_s, m_h, m_{c1}, m_{c2} are the mass of the speaker, HDD, chassis_p1, and chassis_p2 respectively. k_s, k_h, k_c, k_n represent the stiffness of the speaker mounting, HDD mounting, chassis, and notebook base respectively. c_s, c_h, c_c, c_n are the damping coefficient of the speaker mounting, HDD mounting, chassis, and notebook base respectively, f_s is the excitation force of the speaker.

In the analytical model, the transmission route AB is modeled by the speaker mounting with k_s and c_s, the route BC is modeled by the two parts of the chassis with k_c, c_c, k_n, and c_n, and route CD the HDD mounting with k_h and c_h. Furthermore, the whole notebook supported by the ground is also modeled with k_n and c_n.

3.2 Transfer functions

When playing music or other sounds, the oscillating force acted on the diaphragm of the speaker induces the vibration of the speaker body, which subsequently causes the vibration of the chassis and HDD. Transfer function characterizes the vibration transmission route in a simple and clear way. Through Laplace transformation, the above equation (1)-(4) can be solved in frequency domain, which gives the transfer function from speaker to HDD via chassis. Let $Z_i(s) = \wp(z_i(t))$ be the Laplace transfer of $z_i(t)$, the transfer function from excitation force to speaker vibration displacement is

$$H_{AA} = \frac{Z_s}{F_s} = \frac{a_{c1}a_{c2}a_h - a_h b_c^2 - a_{c1}b_h^2}{D_f} \tag{5}$$

and the transfer function of the transmission route AB is

$$H_{AB} = \frac{(a_{c2}a_h - b_h^2)b_s}{D_f} \tag{6}$$

and route BC is

$$H_{BC} = \frac{a_h b_c}{a_{c2}a_h - b_h^2} \tag{7}$$

and route CD is

$$H_{CD} = \frac{b_h}{a_h} \tag{8}$$

where

$$a_s = s^2 m_s + s c_s + k_s$$
$$b_s = s c_s + k_s$$
$$a_{c1} = s^2 m_{c1} + s(c_n + c_s + c_c) + k_n + k_c + k_s$$
$$b_c = s c_c + k_c$$
$$a_{c2} = s^2 m_{c2} + s(c_n + c_h + c_c) + k_n + k_h + k_s$$
$$a_h = s^2 m_h + s c_h + k_h$$
$$b_h = s c_h + k_h$$
$$D_f = a_{c1}a_{c2}a_h - a_h a_s b_c^2 - a_{c1}a_s b_h^2 - a_{c2}a_h b_s^2 + b_h^2 b_s^2$$

Transfer functions (5)-(8) can be combined to represent the transfer function of vibration transmission for all the routes in the notebook PC. For example,

$$H_{AC} = H_{AB}H_{BC} \tag{9}$$

$$H_{AD} = H_{AB}H_{BC}H_{CD} \tag{10}$$

The above functions (9)-(10) describe the vibration transferred from speaker to chassis and HDD.

3.3 Analysis of the vibration transmission in notebook

The analysis of the vibration transmission from speakers to HDD via chassis in the notebook PC is then conducted with the transfer functions. The lumped parameters used in the theoretical model are determined from four dominant modes of HDD vibrations induced by speaker A in the FEM analysis, as shown in Figure 3.

Figure 7 illustrates the analytical results of the vibration frequency response of the speaker, two parts of the chassis and HDD, induced by a unit force acted on the speaker. As shown in Figure 7, there are four peaks corresponding to the four dominant modes at resonant frequencies 281Hz, 492Hz, 712Hz and 1252Hz respectively of the whole system. The frequencies of the four peaks agree well with the FEM results as shown in Figure 3. The vibration energy is firstly transmitted to the speaker, and its vibration spectrum thus has the highest amplitude. The energy is then transmitted to the chassis and finally via the chassis to the HDD. During the transmission, the energy is dissipated by the mounting and chassis structures. Therefore, the amplitude of the vibration is gradually decreased.

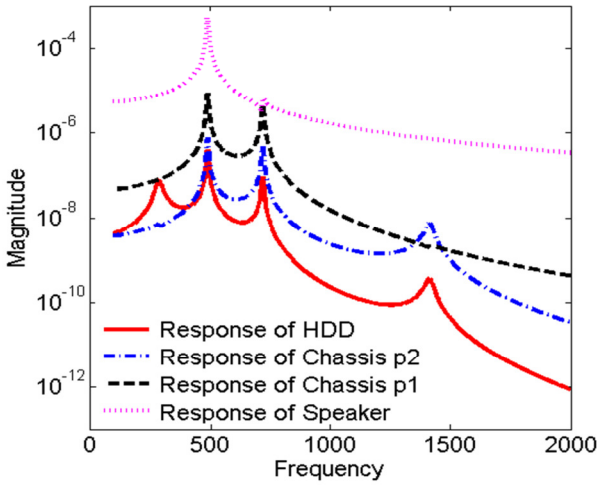


Figure 7. Vibration displacement response of speaker, chassis and HDD

4. Analysis of chassis design and mounting of speaker and HDD

The effects of the chassis design, speaker and HDD mounting on the vibration transmission from speaker to HDD via chassis in the notebook PC are studied, by varying the stiffness and damping of the components in the analytical model.

4.1 Analysis of chassis design

With the transfer functions, the chassis design and components mounting can be analyzed for reducing the vibration transmission energy from speaker to HDD or shifting the resonant mode from a specific frequency range. First of all, chassis can be designed to be less sensitive to the vibration source, and more effective to dissipate vibration energy on the transmission route BC. In the present model, the chassis related stiffness and damping k_c, c_c, k_n, and c_n , can be optimized to reduce the vibration transmission from speaker to HDD, as shown in Figure 8. It is noted that normalized parameters of the stiffness and damping are displayed in this and the subsequent Figures in the parametric studies.

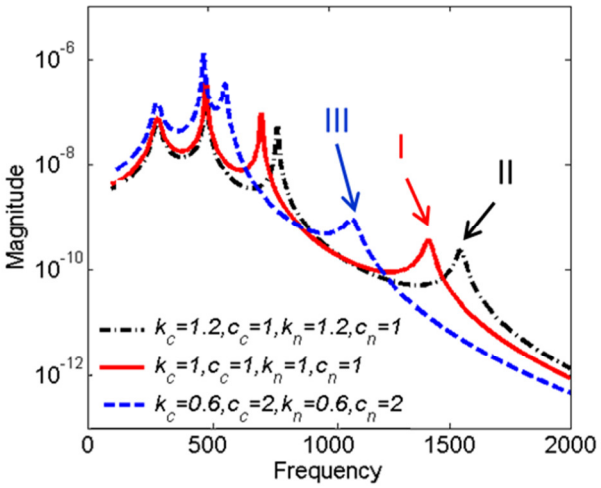


Figure 8. Effect of chassis stiffness and damping on HDD vibration by route BC

It is observed in Figure 8 that by using the chassis designed with smaller stiffness and larger damping, the vibration of transmission at high frequencies above 1000 Hz can be reduced, but the vibration transmission at frequencies below 500 Hz is significantly increased. This phenomenon is characterized in Figure 8 from the solid line I to the dashed line III. It is also seen from the Figure 8 that harder chassis is not good for the vibration transmission at high frequencies above 1000 Hz as observed from the solid line I to the dash-dot line II. As the vibration at low frequency may be compensated by the servo controller [6,7], to reduce the vibration transmission at high frequency, a softer chassis may be helpful. However, if the chassis stiffness is too small, the HDD vibration at low frequency will be increased substantially and out of range of the servo controller.

The relative location between the speaker and HDD in chassis design is one of the key factors relevant to vibration transmission from speaker to HDD by route BC. In the present model, the stiffness and damping change due to the relative locations between speaker and HDD is incorporated into the parameters k_c and c_c of the route BC. In general, if the relative distance between the speaker and HDD is larger, the stiffness between them may be smaller and more vibration energy is dissipated, in case all the other conditions are the same. It is observed from Figure 8 that, with lower k_c and larger c_c , the vibration transmission can be reduced at high frequencies. Therefore, it is recommended to arrange the speakers mounted far away from the HDD in chassis design, such as position #4 and #5 as illustrated in Figure 9.

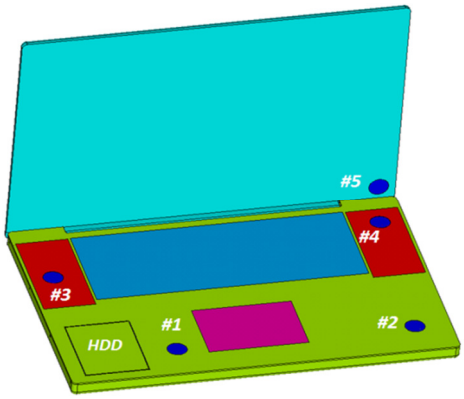


Figure 9. HDD and speaker mounting locations in notebook chassis design

4.2 Analysis of speaker mounting

The influence of the speaker mounting on the vibration of the whole system and respective components is investigated. As stated in the analytical model, the speaker mounting is modeled by the spring and damping elements with stiffness k_s and damping coefficient c_s . Varying these two parameters directly affects the local vibration transmission route AB, as shown in Figure 10, where the stiffness and damping coefficient are normalized to better

understand the effects of these two parameters. The decrease of the stiffness and increase of the damping, make the vibration peak shifting from high frequency to low frequency, and therefore reduces the amplitude at high frequency and amplifies the magnitude at low frequency. The mode shifting characteristics can be used to move the speaker mounting mode away from the frequency of excitation source. The soft speaker mounting with speaker mounting stiffness 1.35×10^5 N/m and damping ratio 5.5%, as illustrated with the dashed line III in Figure 10, it effective to reduce vibrations transmitted to HDD at frequencies above 500 Hz.

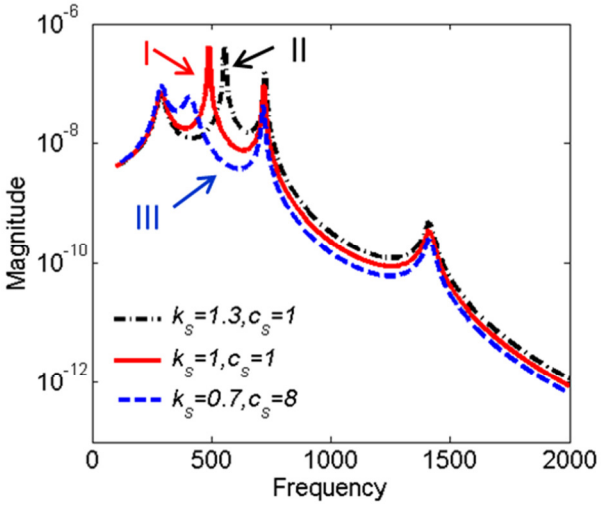


Figure 10. Effect of the different speaker mounting on HDD vibration by route AB

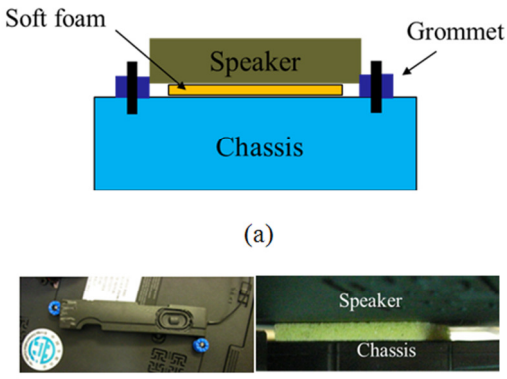


Figure 11. Speaker mounting by foam & rubber grommet

Most of the HDDs are actuated at high frequencies, and the vibration at low frequencies can be compensated by the servo controller optimization [6,7]. Hence it is critical to reduce the amplitude of the vibration transmission from speaker in high frequencies through proper chassis and mounting design. To achieve this, the speaker mounting can be softer, for example, the speaker can be screwed to chassis by using foams and/or rubber grommet, as illustrated in Figure 11. The foams and rubber grommet could help to reduce the stiffness of speaker mounting and increase damping for dissipation of vibration energy.

4.3 Analysis of HDD mounting

HDD mounting has a direct influence on the transmission of vibration to HDD by route CD. A good HDD mounting could isolate the vibration from the build-in speakers. In the present model, k_h and c_h represent the HDD mounting in the notebook PC. The influences of k_h and c_h on the vibration transmitted to HDD induced by speaker are analyzed with the analytical model. The isolation effect of the HDD mounting can be achieved at high frequency by the soft mounting with lower stiffness and higher damping, as demonstrated in Figure 12. By using the soft HDD mounting with the HDD mounting stiffness 2×10^5 N/m and damping ratio 13%, as illustrated with the dashed line III in Figure 12, the HDD vibration at frequencies above 500 Hz is significantly reduced. However, it is also noticed that the first mode is shifted to lower frequency and the transmission amplitude at the frequencies around 200 Hz is increased due to the softer mounting.

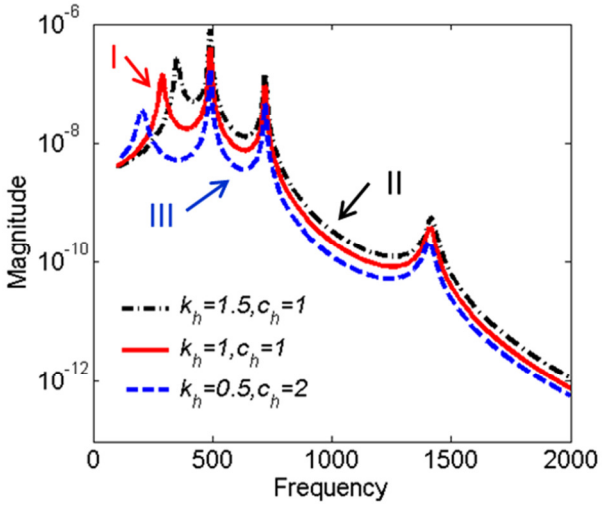
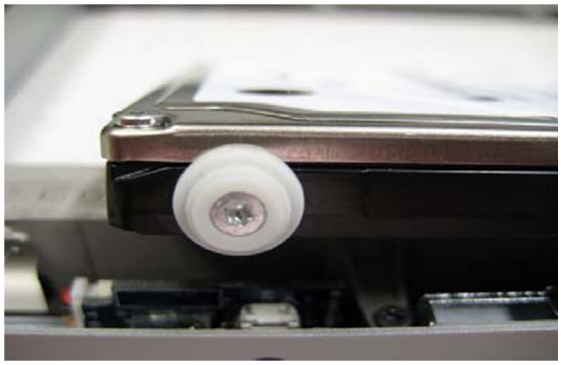
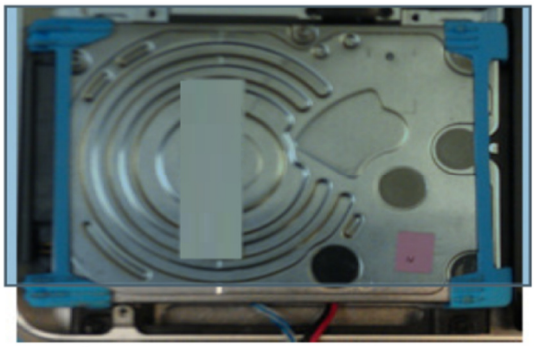


Figure 12. Effect of different HDD mounting on HDD vibration by route CD



(a) HDD mounting with soft supporter



(b) HDD mounting with damper

Figure 13. HDD mounting with soft supporter & damper

From the analysis above, a softer mounting with lower stiffness and higher damping can be used to reduce the vibrations transmission at high frequencies. As shown in Figure 13, HDD mounting to notebook chassis with soft supporters and dampers are effective measures for HDD to intercept vibrations from build-in speakers at high frequencies.

5. Conclusion

The characteristics of HDD vibrations induced by the speakers and the vibration transmission in notebook PC have been studied with FEM and theoretical model. The reduction of vibration transmitted to HDD from build-in speakers via chassis is investigated in three ways including chassis design, speaker mounting and HDD mounting. It is concluded that the illustrated soft speaker mounting (mounting stiffness 1.35×10^5 N/m and damping ratio 5.5%) and soft HDD mounting (mounting stiffness 2×10^5 N/m and damping ratio 13%) are effective for reduction of HDD vibration at frequencies above 500 Hz. Furthermore, a softer chassis with high damping ratio and placing the speaker far away from the HDD is useful for reducing the HDD vibration at high frequencies above 1000 Hz. However, the trade-off of the softer mounting and chassis design is that the vibration of transmission may be increased significantly at frequencies below 500 Hz. Hence optimal design of chassis and components mounting with appropriate stiffness and high damping ratio is essential for reducing the vibration at high frequency and avoiding large vibration at low frequency uncontrollably.

References

[1] G. Ferretti, G. Magnani, and P. Rocco, "Modeling and experimental analysis of the vibrations in hard disk drives," *IEEE/ASME Trans. Mechatronics*, 7(2), pp.152-160, 2002.

[2] S. Lim, Y. B. Chang, N. C. Park, and Y. P. Park, "Optimal design of rubber mounts supporting notebook HDD for shock and vibration isolation," in *Proc. APMRC2006*, Singapore, 2006

[3] Y. Matsuda, S. Nakamura, M. Sega, and Y. Katou, "Flexible support mechanism for hard disk drives to decrease vibration disturbance," *IEEE Trans. Magn.*, 45(11), pp.5108-511, 2009.

[4] T. Semba, M. T. White, and F.Y. Huang, "Adaptive Cancellation of Self-Induced Vibration," *IEEE Trans. Magn.*, 47(7), pp.1958-1963, 2009.

[5] Y. Y. Hu, S. Yoshida_ S. Nakamura, K. Watanabe, W. Z. Lin, E. T. Ong, and J. Q. Mou, "Analysis of built-in speaker-induced structural-acoustic vibration of hard disk drives in Notebook PCs," *IEEE Trans. Magn.,* 45(11), pp.4950-4955, 2009.

[6] T. Semba, M.T. White, and F.Y. Huang, "Adaptive Cancellation of Self-Induced Vibration". *IEEE Trans. Magn.,* 47(7): pp. 1958-1963, 2011.

[7] W. J. Cao, J. Y. Wang, M, Z. Ding, Q. Bi, and K. K. Ooi, "Low Frequency Vibration Detection and Compensation in Hard Disk Drive", *IEEE Trans. Magn.,* 47(7): pp 1964-1969, 2011.

Accurate Throughput Derivation of Pipelined NULL Convention Logic Asynchronous Circuits

Liang Zhou and Scott C. Smith
Department of Electrical Engineering
University of Arkansas
Fayetteville, Arkansas, USA
kingdom701@gmail.com and smithsco@uark.edu

Abstract - *A throughput estimation formula for optimally pipelining NULL Convention Logic (NCL) asynchronous circuits was presented in the literature. However, it ignores register delays. This paper presents a precise throughput derivation formula for pipelined NCL circuits. The formula was verified by Spice simulation and can be used for static timing analysis.*

Keywords: asynchronous circuits; pipelining; throughput derivation; NULL Convention Logic (NCL)

I. INTRODUCTION

The development of synchronous circuits currently dominates the semiconductor design industry. However, there are major limiting factors to the synchronous, clocked approach, including the increasing difficulty of clock distribution, increasing clock rates, decreasing feature size, increasing power consumption, timing closure effort, and difficulty with design reuse. Asynchronous (clockless) circuits require less power, generate less noise, produce less electro-magnetic interference (EMI), and allow for easier reuse of components, compared to their synchronous counterparts.

Quasi-delay-insensitive (QDI) NULL Convention Logic (NCL) [1] is one of the primary delay-insensitive asynchronous paradigms; a number of circuits have been successfully designed using NCL [2]. NCL systems can be optimized for speed by partitioning the combinational circuitry and inserting delay-insensitive (DI) registers and corresponding completion components, utilizing either the full-word or bit-wise completion strategy [3].

A throughput estimation formula for pipelined NCL systems was proposed in the literature [3]. However, it ignores register delays and is therefore not precise, especially when applied to finely pipelined NCL systems. As a result, it cannot be used for static timing analysis. In this paper, we propose an accurate throughput derivation formula including register delays for pipelined NCL systems. The formula was verified by Spice simulation and can be used for static timing analysis.

Section II presents an overview of NCL. Section III derives the throughput of non-pipelined NCL systems; Section IV derives the throughput of pipelined NCL systems. Section V verifies the formulas utilizing Spice simulation; and Section VI provides conclusions and future work.

II. INTRODUCTION TO NCL

NCL circuits utilize multi-rail logic, such as dual-rail, to achieve delay-insensitivity. A dual-rail signal, *D*, consists of two wires, D^0 and D^1, which may assume any value from the set {DATA0, DATA1, NULL}. The DATA0 state (D^0 = 1, D^1 = 0) corresponds to a Boolean logic 0, the DATA1 state (D^0 = 0, D^1 = 1) corresponds to a Boolean logic 1, and the NULL state (D^0 = 0, D^1 = 0) corresponds to the empty set meaning that the value of *D* is not yet available. The two rails are mutually exclusive, such that both rails can never be asserted simultaneously; this state is defined as an illegal state.

NCL circuits are comprised of 27 fundamental gates [4]. These 27 gates constitute the set of all functions consisting of four or fewer variables. The primary type of threshold gate, shown in Fig. 1, is the *THmn* gate, where $1 \leq m \leq n$. *THmn* gates have *n* inputs. At least *m* of the *n* inputs must be asserted before the output will become asserted. NCL threshold gates are designed with *hysteresis* state-holding capability such that all asserted inputs must be de-asserted before the output will be de-asserted. Therefore, a THnn gate is equivalent to an n-input C-element [5] and a TH1n gate is equivalent to an n-input OR gate. NCL threshold gates may also include a *reset* input to initialize the output. Circuit diagrams designate resettable gates by either a *d* or an *n* appearing inside the gate, along with the gate's threshold. *d* denotes the gate as being reset to logic 1; *n*, to logic 0. The static NCL threshold gate is shown in Fig. 2, where hold0 = set' and hold1 = reset'.

NCL circuits communicate using request and acknowledge signals, *Ki* and *Ko*, respectively, to prevent the current DATA wavefront from overwriting the previous DATA wavefront, by ensuring that the two DATA wavefronts are always separated by a NULL wavefront. The acknowledge signal from the receiving circuit is the request signal to the sending circuit. When the receiver circuit latches the input DATA, the corresponding *Ko* signal will become logic 0, indicating a Request-For-NULL (RFN); and when it latches the input NULL, the corresponding *Ko* signal will become logic 1, indicating a Request-For-DATA (RFD). When the sending circuit receives a RFD/RFN on its *Ki* input, it will allow a DATA/NULL wavefront to be output, respectively. Request/acknowledge signals are usually generated using C-elements.

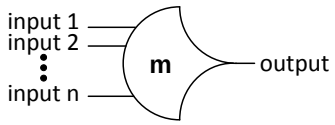


Figure 1. THmn threshold gate.

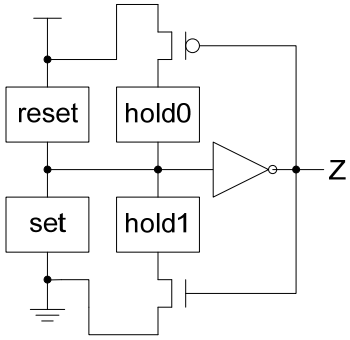


Figure 2. Static NCL threshold gate design.

III. Throughput Derivation of Non-Pipelined NCL Systems

Non-pipelined NCL systems contain two DI registers, one at both the input and at the output. In NCL systems, the DATA-to-DATA cycle time (TDD) [3] has an analogous role to the clock period in a synchronous system and is the reciprocal of throughput. To derive the throughput of NCL systems, the external interface is assumed to respond to *Ko* signals and circuit outputs immediately. Four system parameters are defined in the handshaking sequence, as shown in Figs. 3-6:

TD: DATA propagation time (from when all input register inputs become DATA and *Ki* signals become RFD, to when all inputs of the output register become DATA).

TRFN: NULL request time (from when all output register inputs become DATA and *Ki* signals become RFD, to when all *Ki* inputs of the input register become RFN).

TN: NULL propagation time (from when all input register inputs become NULL and *Ki* signals become RFN, to when all inputs of the output register become NULL).

TRFD: DATA request time (from when all output register inputs become NULL and *Ki* signals become RFN, to when all *Ki* inputs of the input register become RFD).

In non-pipelined NCL systems, the end of the DATA request time of the current cycle is the beginning of the DATA propagation time of next cycle. Therefore, TDD = TD + TRFN + TN + TRFD and throughput = 1 / TDD. The estimation in [3] is inaccurate as it ignores register delays in the definition of the four system parameters, since it was used to determine where to add pipeline stages, not to precisely calculate timing.

If a gate delay is used as the minimal delay unit, then TD = TN, TRFN = TRFD, and the formula can be simplified. The schematic of a 1-bit DI register is shown in Fig. 7. The gate delay from its input to output is 1. The gate delay from its input to *Ko* is 1.5, because the inverted TH12 gate is equivalent to a NOR gate, which doesn't have an output inverter, so has 0.5 gate delay compared to other types of threshold gates. As shown in Fig. 8, if the Combinational Logic has TCOMB gate delays and the Completion Logic has TCOMP gate delays, then TD = TN = 1 + TCOMB, TRFN = TRFD = 1.5 + TCOMP, and TDD = 2 * (2.5 + TCOMB + TCOMP). The formula in [3] estimates TDD as 2 * (TCOMB + TCOMP), which is imprecise.

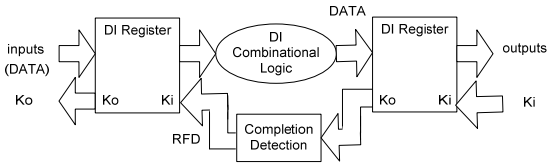


Figure 3. DATA propagation time.

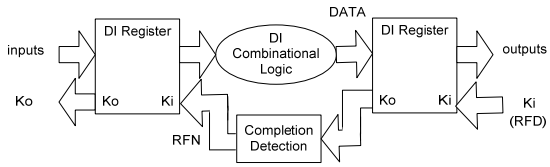


Figure 4. NULL request time.

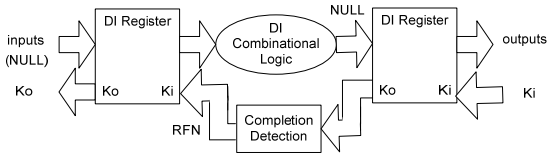


Figure 5. NULL propagation time.

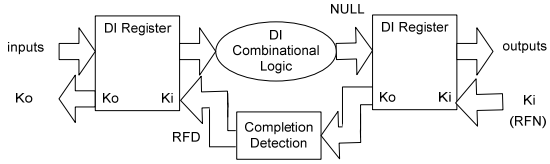


Figure 6. DATA request time.

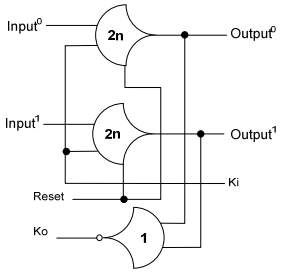


Figure 7. 1-bit DI register

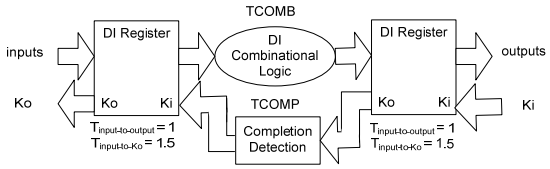


Figure 8. System parameters of non-pipelined NCL systems using gate delays.

IV. Throughput Derivation of Pipelined NCL Systems

The throughput of pipelined NCL systems is determined by the stage with the largest TDD. In this paper, we define a stage as shown in Fig. 9, where registers are shared by adjacent stages. TDD_i of $stage_i$ has 3 possibilities [3]. As shown in Fig. 10, if $TN_i + TRFD_i < TRFD_{i-1} + TD_{i-1}$, then when all of the *Ki* signals of the input register of $stage_i$ become RFD, it has to wait for $(TRFD_{i-1} + TD_{i-1} - TN_i - TRFD_i)$ before all of the inputs of the input register of $stage_i$ become DATA. Therefore, $TDD_i = TD_i + TN_i + TRFD_i + TRFN_i + (TRFD_{i-1} + TD_{i-1} - TN_i - TRFD_i) = TRFD_{i-1} + TD_{i-1} + TD_i + TRFN_i$. Similarly, as shown in Fig. 11, if $TD_i + TRFN_i < TRFN_{i-1} + TN_{i-1}$, then when all of the Ki signals of the input register of $stage_i$ become RFN, it has to wait for $(TRFN_{i-1} + TN_{i-1} - TD_i - TRFN_i)$ before all of the inputs of the input register of $stage_i$ become NULL. Therefore, $TDD_i = TD_i + TN_i + TRFD_i + TRFN_i + (TRFN_{i-1} + TN_{i-1} - TD_i - TRFN_i) = TRFN_{i-1} + TN_{i-1} + TN_i + TRFD_i$. If none of the above two conditions are true, $TDD_i = TD_i + TN_i + TRFD_i + TRFN_i$.

In summary, TDD = max $(TRFD_{i-1} + TD_{i-1} + TD_i + TRFN_i$, $TRFN_{i-1} + TN_{i-1} + TN_i + TRFD_i$, $TD_i + TN_i + TRFD_i + TRFN_i)$ for *i* in all of the stages. If gate delay is used as the minimal delay unit, then TD = TN, TRFN = TRFD, and the formula can be simplified as TDD = max $(2 * (2.5 + TCOMB_i + TCOMP_i))$ for *i* in all of the stages. The formula in [3] estimates TDD as $2 * (TCOMB_i + TCOMP_i)$, which is imprecise, especially when applied to finely pipelined NCL systems.

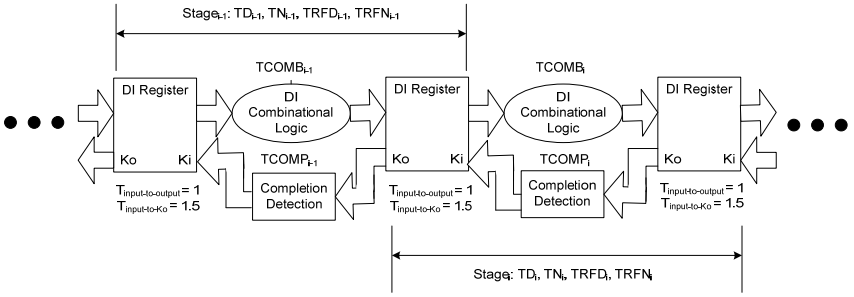


Figure 9. System parameters of pipelined NCL systems with gate delay.

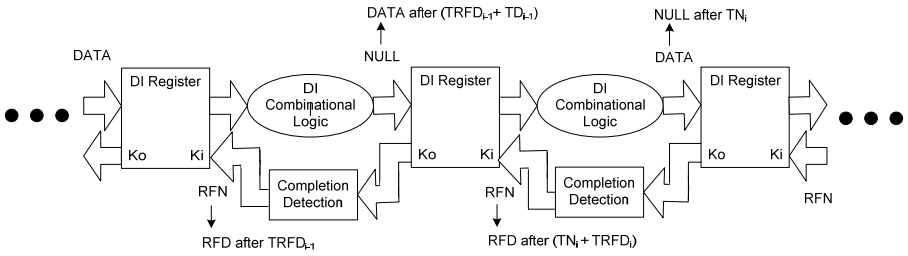


Figure 10. TDD_i derivation when $TN_i + TRFD_i < TRFD_{i-1} + TD_{i-1}$.

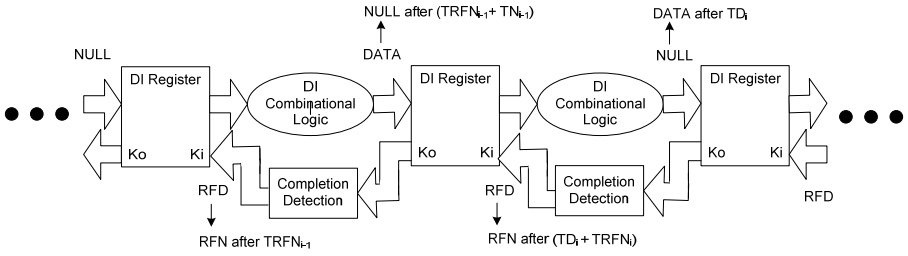


Figure 11. TDD_i derivation when $TD_i + TRFN_i < TRFN_{i-1} + TN_{i-1}$.

V. SPICE VERIFICATION OF THE FORMULAS

To verify the formula proposed in Section IV, a 3-stage full-word pipelined 4×4 NCL multiplier [3] was implemented with IBM cmos10lpe 65nm process at the transistor level and simulated with Cadence Spectre simulator. A VerilogA controller was designed to characterize the four system parameters of each stage and to measure TDD with different input patterns. As shown in Table I, the measured TDD was compared with predicted TDD calculated from the formula and they matched.

VI. CONCLUSION

In this paper, we propose an accurate throughput derivation formula for pipelined NCL systems. The formula was verified by Spice simulation and can be used for static timing analysis. Future work consists of NCL threshold gate library characterization with Synopsys NCX and static timing analysis with Synopsys PrimeTime.

REFERENCES

[1] K. M. Fant and S. A. Brandt, "NULL Convention Logic: A Complete and Consistent Logic for Asynchronous Digital Circuit Synthesis," *International Conference on Application Specific Systems, Architectures, and Processors*, pp. 261-273, 1996.

[2] R. Jorgenson, L. Sorensen, D. Leet, M. Hagedorn, D. Lamb, T. Friddell, and W. Snapp,, "Ultralow-power operation in subthreshold regimes applying clockless logic," *Proceedings of the IEEE*, vol. 98, Feb. 2010, pp. 299–314.

[3] S. C. Smith, R. F. DeMara, J. S. Yuan, M. Hagedorn, and D. Ferguson, "Delay-Insensitive Gate-Level Pipelining," *Elsevier's Integration, the VLSI Journal*, Vol. 30/2, pp. 103-131, October 2001.

[4] Gerald E. Sobelman and Karl M. Fant, "CMOS Circuit Design of Threshold Gates with Hysteresis," *IEEE International Symposium on Circuits and Systems*, pp. 61-65, 1998.

[5] D. E. Muller, "Asynchronous Logics and Application to Information Processing," in *Switching Theory in Space Technology*, Stanford University Press, pp. 289- 297, 1963.

TABLE I. SPICE VERIFICATION RESULTS

System Parameters and Derivations (ps)	Input Patterns		
	X = 15, Y = 15	***X = 0, Y = 0***	***X = 7, Y = 8***
TD1	267	279	349
TN1	393	500	658
TRFD1	346	358	357
TRFN1	413	412	413
TD2	261	249	320
TN2	404	420	424
TRFD2	336	336	345
TRFN2	392	391	363
TD3	251	249	226
TN3	370	530	246
TRFD3	316	310	309
TRFN3	303	303	301
TD1 + TN1 + TRFD1 + TRFN1	1420	1551	1778
TRFD1 + TD1 + TD2 + TRFN2	1267	1279	1390
TRFN1 + TN1 + TN2 + TRFD2	1548	1669	1842
TD2 + TN2 + TRFD2 + TRFN2	1395	1397	1454
TRFD2 + TD2 + TD3 + TRFN3	1153	1138	1193
TRFN2 + TN2 + TN3 + TRFD3	1484	1652	1344
TD3 + TN3 + TRFD3 + TRFN3	1242	1393	1083
Predicted TDD	1548	1669	1842
Measured TDD	1548	1669	1842

Effect of Channel Lengthening and Threshold Voltage Variation on a Nanometric Gate's Delay and Power

Azam Beg[1], **Amr Elchouemi**[2], **and Raahim Beg**[3]
[1]United Arab Emirates University, Faculty of Information Technology, Al-Ain, United Arab Emirates
[2]Hewlett Packard, Austin, TX, USA
[3]Liwa School, Al-Ain, United Arab Emirates

Abstract - *Rapid scaling of CMOS devices in the recent years has not only increased the leakage power consumption but also increased the susceptibility of the circuits to device parameter variations. As a method for mitigating these effects, we have investigated the use of MOS transistors with longer-than-minimum channels. We performed Monte Carlo simulations to quantify the effects of threshold voltage variation and channel lengthening. With increased channel lengths, we were able to attain reduced susceptibility to the variations. Power reduction was also achieved but at the cost of performance. The longer-than-minimum channels technique could be useful for low power and mobile applications.*

Keywords: Low power circuits, parameter variation, threshold voltage, channel length, performance

1 Introduction

In the last few decades, continual reduction of transistor dimensions has resulted in a substantial increase in the leakage power's share in the total power dissipated by the CMOS circuits. So controlling the leakage current is an obvious approach for reducing the power consumption. Yet another adverse result of transistor scaling is an increase of parameter fluctuations, which causes a loss of both the performance and the reliability. The variations occur during the fabrication process or appear later due to aging. With feature scaling, it is getting harder to retain a consistent value of threshold voltage (V_{TH}) for a large number of transistors on an IC. Deviations in V_{TH} occur mainly due to randomness of location of and the number of dopant atoms. The results in [1] show that V_{TH} is approximately normally distributed with a standard deviation of:

$$\sigma_{V_{TH}} \approx 3.19\times10^{-8}\cdot\frac{t_{ox}\times N_A^{0.4}}{\sqrt{L_{eff}\times W_{eff}}} \qquad (1)$$

where t_{ox} is the oxide thickness, N_A is the channel doping, W_{eff} is the channel width, and L_{eff} is the channel length.

As we can see in equation (1), the variation in V_{TH} can be reduced by increasing L and/or W. Normally, the VLSI designers start with minimum L_{nMOS} and L_{pMOS}, and $W_{nMOS} = 2\times L_{nMOS}$. Then they adjust W_{pMOS} to balance the high-to-low and low-to-high propagation delays. Doing so increases the area of pMOS transistor and makes it even more reliable than nMOS which is less reliable than the pMOS to begin with. This means that despite increased area, there in no improvement in overall gate's reliability [2]. In this paper, we present a transistor sizing scheme in which we retain the traditional W_{pMOS} and W_{nMOS} ratios but lengthen L_{pMOS} and L_{nMOS} dimensions (while keeping W's constant) for a CMOS inverter. With simulations, we have measured the sensitivity of delay and power to the channel length. We also performed Monte Carlo (MC) simulations with randomly distributed V_{TH}, and analyzed its effect on normalized values of delay and power.

2 Related Work

2.1 Leakage Power Reduction

The topic of leakage power estimation and reduction has been extensively researched in the past 10-15 years. A brief review of the work is given below.

Halter and Najm [3] proposed a method for placing circuits into *low leakage standby state* and reported power reduction of 7%–54% for several ISCAS-89 benchmark circuits. Yasuda and Hosokwa [4] reduced standby leakage current by using low V_{TH} devices for high performance and low power dissipation in active and standby modes. Ishibashi *et al*. [5] presented a *self-adjusting forward body bias* technique for maximizing forward body bias voltage without increasing the forward current. Bol *et al.* [6] introduced an ultra-low power logic which exhibited low leakage currents but operated at very low frequency. The logic uses extra transistors to create a feedback mechanism to cut-off current in the pMOS-nMOS stack. Calhoun [7] introduced the idea of *ultradynamic voltage scaling* (UDVS) wherein the supply voltage swung between above-threshold and sub-threshold levels as needed by the operation being performed. For the

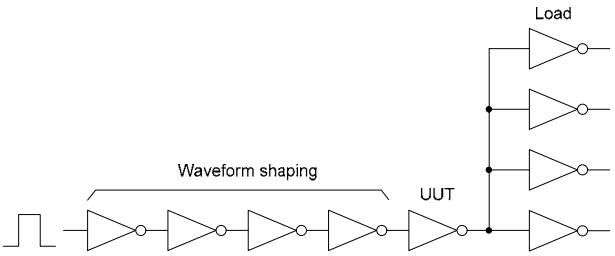


Figure 1. Test circuit of an inverter with fan-out of 4.

purpose of increasing reliability and for reducing power, reversing of transistor dimensions in CMOS gates was proposed in [8]; this way, the device area remains the same but there is an appreciable increase in the delay time [9].

2.2 Parameter Variation

The effects of parameter variations have been a subject of many research works some of which are mentioned here.

Mukhopadhyay and Roy [10] presented analytical models for estimating variation effects in 50 nm CMOS devices. Taylor and Fortes [11] showed how changes in V_{TH} affected the output voltage levels of an inverter and a NAND-2 gate. Agarwal and Nassif [12] presented a detailed study of techniques for parameter measurement on actual Silicon (65 nm SOI), in order to correlate analytical models with the actual hardware. Asenov [13] studied the impact of statistical variation in MOS transistors due to random dopants (location and count), line edge roughness, polysilicon granularity and oxide thickness changeability. They concluded that random dopants were the most dominant parameter affecting the device scaling. Ibrahim [14] studied the effect of V_{TH} variation on a NAND2-CMOS gates using Bayesian networks. However, no actual simulations were used to measure power or delay. Remarsu and Kundu [15] characterized the process variation in thermal sensors operating at different temperatures. Tang *et al.* [16] incorporated a simplistic transistor model in a statistical simulation engine for the purpose of estimating the delay, power, and noise effects.

3 Experiments and Results

In this paper, we have analyzed a CMOS inverter using NG-Spice ver. 22 and a chain circuit (see Figure 1). We used 16 nm high performance (HP) v2.1, metal gate, high-*k*, and strained-Si predictive technology model (PTM) [17]. A square-wave (50% duty-cycle) of 10 MHz was used as the stimuli. The same frequency was used for both above- and below-threshold voltage experiments.

Using simulations, we studied the effect of increased channel length L on delay and power under different conditions, i.e., a range of V_{DD} with constant V_{TH}, and above and sub-threshold V_{DD} while V_{TH} changes randomly. The channel lengths used are:

$$\{L \mid L_{min}, 1.2\times L_{min}, 1.4\times L_{min}, 1.6\times L_{min}\}\text{an.}$$

In Figure 2, we show the sensitivity of inverter delay to V_{DD}. We notice that at sub-threshold V_{DD} of 200 mV, the delay penalty is the highest among all lengths. The penalty drops as V_{DD} increases to 1 V. Increase in delays ranges between 1. 7× and 17.5×.

Figure 3 shows the relationship of V_{DD} and power. Expectedly, low V_{DD} results in low power consumption and

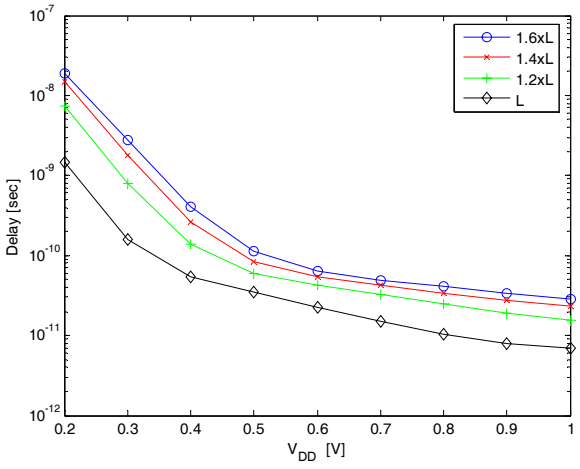


Figure 2. Relationship of an inverter's delay to supply voltage with different channel lengths

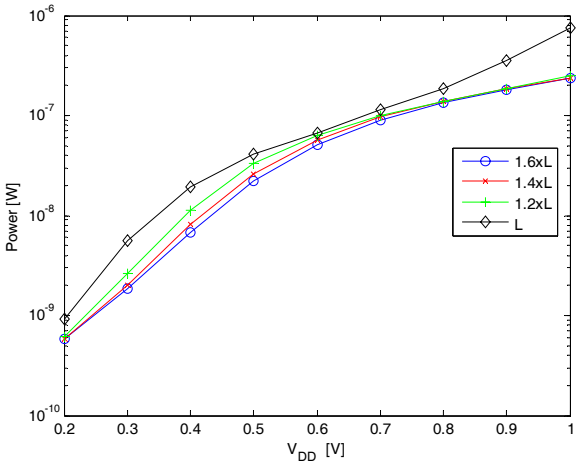


Figure 3. Relationship of an inverter's power to supply voltage with different channel lengths

vice versa. At nominal voltage of 700 mV, increase in L has the lowest benefit, while at highest supply voltage of 1 V, going to $1.2\times L_{min}$ provides 3× savings in power. Overall, the savings in power range between just 1.05× and 3.1×. Besides, supply voltage, the drop in power consumption can be attributed to decrease in leakage current due to increased L.

We looked into changes in delay and power for an inverter (unit-under-test in Figure 1) when it was subject to V_{TH} variations using 1000 Monte Carlo simulations. Mean values of V_{TH} for nMOS and pMOS transistors were taken from 16 nm PTM models [17], and the standard deviation for V_{TH} was based on equation (1).

The histograms for delay and power for V_{DD} = 300 mV are shown in Figures 4 and 5, respectively; and for V_{DD} = 700 mV, in Figures 6 and 7, respectively. As expected, the delays get longer as L increases.

Table 1 lists the normalized delay values (calculated by $\mu[delay_i]/\sigma[delay_i]$). The smallest delay is seen with $L = 1.2\times L_{min}$ when V_{DD} = 300 mV, and for $L = L_{min}$ when V_{DD} = 700 mV. In Table 2, we notice that the lowest power consumption (calculated by $\mu[power_i]/\sigma[power_i]$) happens at V_{DD} = 300 mV when $L = 1.2\times L_{min}$, while V_{DD} = 700 mV

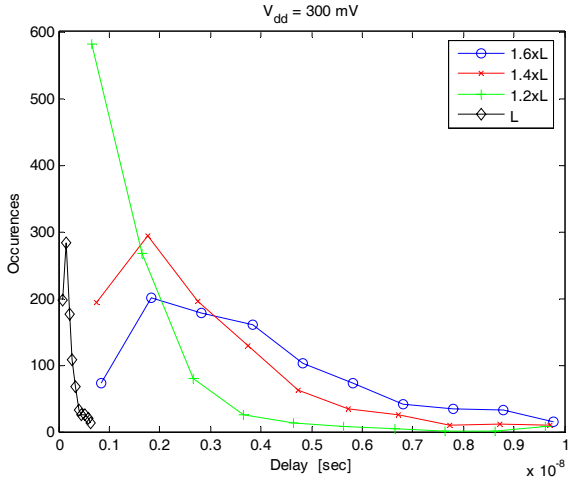


Figure 4. Delay histogram for different channel lengths while operating with sub-threshold supply voltage

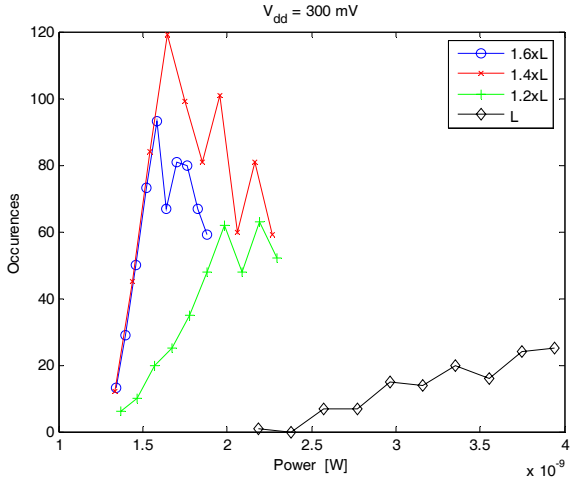


Figure 5. Power histogram for different channel lengths while operating with sub-threshold supply voltage

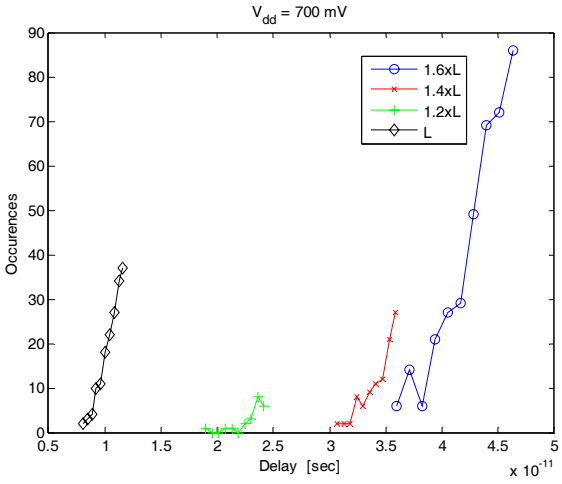


Figure 6. Delay histogram for different channel lengths while operating with nominal supply voltage

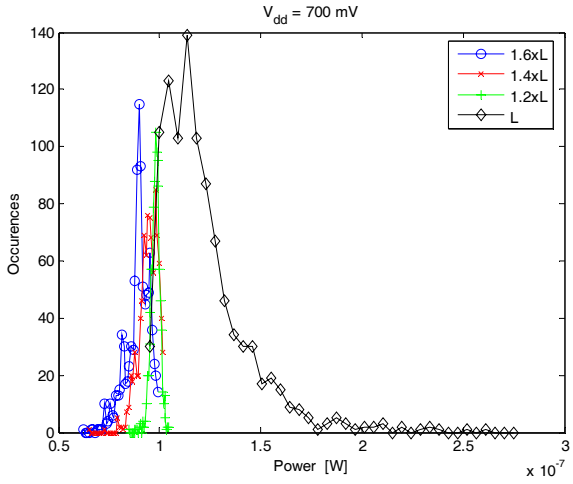


Figure 7. Power histogram for different channel lengths while operating with nominal supply voltage

gives the lowest power with $L = L_{min}$. These results confirmed our expectation that the longer channels decrease both power consumption and the performance.

Lastly, we investigated how larger deviations in V_{TH} reduce the noise margin. In Figure 8, we can see that the output levels drop when V_{TH} is lower than its nominal value. V_{out} for a normally-sized transistor (with length $L = L_{min}$) has the worst decline. But with just 20% increase in L (i.e., $1.2 \times L_{min}$) the output voltage level improves significantly. However, V_{out} in all cases shown in the figure stays above $0.9 \times V_{DD}$ = 630 mV. In comparison, when the input is a logic '1', V_{out} is the worst when $L = L_{min}$ and when V_{TH} is 20% of the normal value; in this case the drop in V_{out} exceeds the 10% noise margin (see Figure 9). Nevertheless, lengthening the channel by 20% makes appreciable improvement in the noise margin even with the lowest V_{TH} (20% of normal).

4 Conclusions and Ongoing Work

In this paper, we have shown increased-channel-lengths ($L > L_{min}$) as a technique for reducing the adverse effect of V_{TH} variation on a CMOS inverter's behavior. With longer channels, reduction in power is also achieved albeit at the cost of performance and area.

Our continued work involves using $L > L_{min}$ for other primitive gates and simple circuits (for example, a 28-transistor one-bit full adder). We are also studying the variation of other parameters, such as line edge roughness, temperature, etc.

5 Acknowledgments

We are thankful to Dr Borivoje Nikolic, University of Berkeley, CA, USA for his valuable inputs on the research topic and the subsequent feedback on the initial version of this paper.

6 References

[1] Li, Y., Hwang, C.-H., Yeh, T.-C. and T.-Y. Li. 2008. Large-scale atomistic approach to random-dopant-induced

Table 1. Normalized delay with random V_{TH} variations

Channel length L	Normalized delay with V_{DD} = 300 mV	Normalized delay with V_{DD} = 700 mV
L_{min}	0.99	**4.74**
$1.2 \times L_{min}$	**0.64**	7.69
$1.4 \times L_{min}$	0.97	8.89
$1.6 \times L_{min}$	1.02	8.06

Table 2. Normalized power with random V_{TH} variations

Channel length L	Normalized power with V_{DD} = 300 mV	Normalized power with V_{DD} = 700 mV
L_{min}	3.83	**5.14**
$1.2 \times L_{min}$	**3.34**	42.42
$1.4 \times L_{min}$	3.81	18.34
$1.6 \times L_{min}$	4.72	14.42

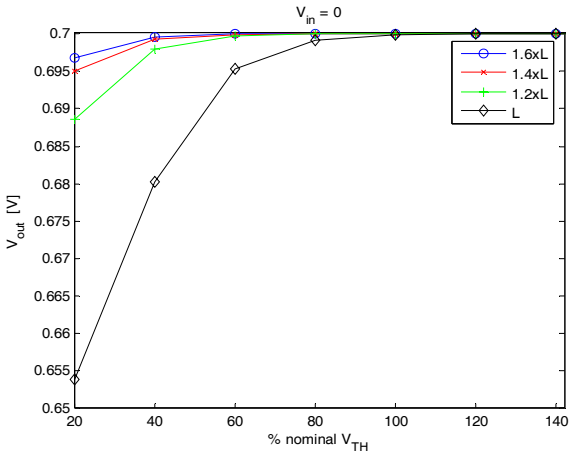


Figure 8. Output voltage level's (V_{out}) sensitivity to V_{TH} when V_{in} = logic '0'

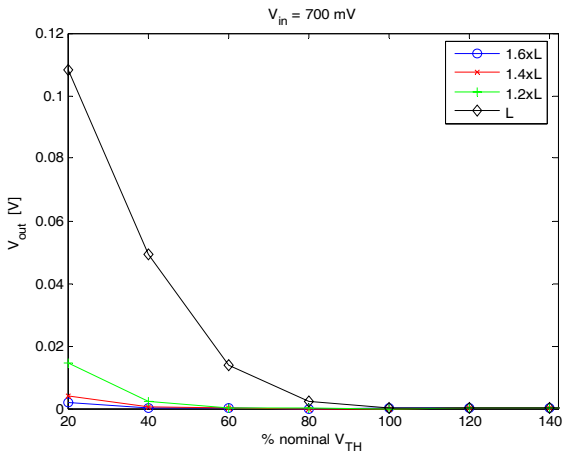


Figure 9. Output voltage level's (V_{out}) sensitivity to V_{TH} when V_{in} = logic '1'

characteristic variability in nanoscale CMOS digital and high-frequency integrated circuits. In *Proceedings of the Int. Conf. CAD (ICCAD)* (San Jose, CA, USA, Nov. 2008), 278–285.

[2] Ibrahim, W. and Beiu, V. 2011. Devices and input vectors are shaping von Neumann multiplexing. *IEEE Trans. Nanotech.*, 10 (May 2011), 606–616.

[3] Halter, J. P. and Najm, F. N. 1997. A gate-level leakage power reduction method for ultra-low-power CMOS circuits. In *Proceedings of the IEEE 1997 Custom Integrated Circ. Conf.* (Santa Clara, CA, USA, May 1997), 475–478.

[4] Yasuda, T. and Hosokawa, K. 2003. A low power CMOS circuit with variable source scheme (VSCMOS). In *Proceedings of the Asia & South Pacific Des. Automation Conf. (ASP-DAC 2003)* (Kitakyushu, Japan, Jan. 2003), 404–407.

[5] Ishibashi, K., Fujimoto, T., Yamashita, T., Okada, H., Arima, Y., Hashimoto, Y., Sakata, K., Minematsu, I., Itoh, Y., Toda, H., Ichihashi, M., Komatsu, Y., Hagiwara, M. and Tsukada, T. 2006. Low-voltage and low-power logic, memory, and analog circuit techniques for SoCs using 90 nm technology and beyond. *IEICE Trans. Electron.*, E89–C, 3 (Mar. 2006), 250–262.

[6] Bol, D., Vos, J. D., Flandre, D. and Legat, J.-D. 2009. Ultra-low-power high-noise-margin logic with undoped FD SOI devices. In *Proceedings of the IEEE Int. SOI Conf.* (Foster City, CA, USA, Oct. 2009), 97–98.

[7] Calhoun, B. H., Ryan, J. F., Khanna, S., Putic, M. and Lach, J. 2010. Flexible circuits and architectures for ultralow power. *Proc. IEEE*, 98 (Feb. 2010), 253–266.

[8] Sulieman, M. H., Beiu, V. and Ibrahim, W. 2010. Low-power and highly reliable logic gates: Transistor-level optimizations. In *Proceedings of the Int. Conf. IEEE-NANO* (Seoul, Korea, Aug. 2010), 254–257.

[9] Beg, A., Beiu, V. and Ibrahim, W. 2011. Atto Joule CMOS gates using reversed sizing and W/L swapping. In *Proceedings of the IEEE 9th Int. New Circ. & Syst. Conf. (NEWCAS 2011)* (Bordeaux, France, Jun. 2011), 498–501.

[10] Mukhopadhyay, S. and Roy, K. 2003. Modeling and estimation of total leakage current in nano-scaled CMOS devices considering the effect of parameter variation. In *Proceedings of the 2003 Int. Symp. Low Power Electronics & Des. (ISLPED'03)* (Seoul, Korea, Aug. 2003). ACM, 172–175.

[11] Taylor, E. and Fortes, J. 2005. Device variability impact on logic gate failure rates. In *Proceedings of the 16th IEEE Int. Conf. Application-Specific Syst., Archit. & Processors (ASAP 2005)*Jul. 2005), 247–253.

[12] Agarwal, K. and Nassif, S. 2007. Characterizing process variation in nanometer CMOS. In *Proceedings of the 44th Annual Des. Automation Conf.* (San Diego, CA, USA, 2007). ACM, 396–399.

[13] Asenov, A. 2007. Simulation of statistical variability in nano MOSFETs. In *Proceedings of the 2007 IEEE Symp. VLSI Tech.*Jun. 2007), 86–87.

[14] Ibrahim, W. and Beiu, V. 2009. Reliability of NAND-2 CMOS gates from threshold voltage variations. In *Proceedings of the 6th Int. Conf. Innovations in Information Tech. (IIT'09)* (Al-Ain, United Arab Emirates, Mar. 2009), 310–314.

[15] Remarsu, S. and Kundu, S. 2009. On process variation tolerant low cost thermal sensor design in 32nm CMOS technology. In *Proceedings of the 19th ACM Great Lakes Symp. VLSI* (Boston Area, MA, USA, May 2009). ACM, 487–492.

[16] Tang, Q., Zjajo, A., Berkelaar, M. and Meijs, N. v. d. 2010. Transistor-level gate modeling for nano CMOS circuit verification considering statistical process variations. In *Proceedings of the 20th Int. Conf. Integr. Circ. & Syst. Design: Power and Timing Modeling, Optimization and Simulation (PATMOS'10)* (Grenoble, France, 2010). Springer-Verlag, 190–199.

[17] Zhao, W. and Cao, Y. 2006. New generation of predictive technology modeling for sub-45nm early design exploration. *IEEE Trans. Electr. Dev.*, 53 (Nov. 2006), 2816–2823.

SESSION

OPERATING SYSTEMS TOOLS AND DESIGN + COMMUNICATION SYSTEMS AND DESIGN

Chair(s)

TBA

A Survey on Computer System Memory Management

Qi Zhu
Department of Computer Science, University of Houston - Victoria, Victoria, Texas, USA

Abstract - *Computer memory is central to the operation of a modern computer system; it stores data or program instructions on a temporary or permanent basis for use in a computer. In this paper, various memory management and optimization techniques to improve computer performance are reviewed, such as the hardware design of the memory organization, the memory management algorithms and optimization techniques, and some hardware and software memory optimization techniques.*

Keywords: ***Memory Organization, Management, Optimization***

1 Introduction

Computer memory system, including ROM, RAM, Cache, virtual memory, is a necessity for any modern computer system. The notation on computer memory usually refers to main memory or primary memory, which temporarily holds the data and instructions needed in process execution by the Central Processing Unit (CPU). A perfect memory organization is one that has unlimited space and is infinitely fast so that it does not limit the processor, which is not practically implementable such as Universal Turing Machine [56]. In reality, there is an increasing gap between the speed of memory and the speed of Microprocessors [24]. In the early 1980s, the average access time of memory was about 200 ns, which was nearly the same as the clock cycle of the commonly used 4.77 MHz (210 ns) microprocessors of the same period. Three decades later, the typical speed of a home microprocessor is 4 GHz (0.25 ns), however, memory access time is staying around 10 nanoseconds. Thus, the growing processor - memory performance gap becomes the primary obstacle to improve computer system performance [32].

This paper surveys computer memory management strategies and various memory optimization techniques in order to improve computer system performance.

2 Memory Architecture Designs

In this section, we start by looking at the design of the memory organization in modern computer systems to facilitate the performance optimization.

2.1 Memory hierarchical structure

Memory was just a single-level scheme for early computers. However, while computers became faster and computer programs were getting bigger, especially multiple processes that were concurrently executed under the same computer system, a single main memory that was both fast enough and large enough had not really been available. As a result, memory hierarchical structure was designed to provide with rather fast speed at low cost considering of economic and physical constraints [6]. Because of the cost-performance trade-offs, the memory hierarchy of modern computers usually contains registers, cache, main memory, and virtual memory. The concept of memory hierarchy takes advantage of the principle of locality [14], which states that accessed memory words will be referenced again quickly (temporal locality) and that memory words adjacent to an accessed word will be accessed soon after the access in question (spatial locality). Loops, functions, procedures and variables used for counting and totaling all involve temporal locality, where recently referenced memory locations are likely to be referenced again in the near future. Array traversal, sequential code execution, modules, and the tendency of programmers (or compilers) to place related variable definitions near one another all involve spatial locality - they all tend to generate clustered memory references.

2.2 Cache

In modern computer systems, caches are small, high-speed, temporary storage areas to hold data duplicating original values stored in main memory which are currently in use [24], where the original data is expensive to access compared to the cost of accessing from the cache. When a CPU needs a particular piece of information, it first checks whether it is in the cache. If it is, the CPU uses the information directly from the cache; if it is not, the CPU uses the information from the memory, putting a copy in the cache under the assumption that it will be needed again soon. Information located in cache memory is accessed in much less time compared to that located in memory. Thus, a CPU spends far less time waiting for instructions and data to be fetched and/or stored.

In this section, some cache optimization techniques are discussed to escalate the cache performance through improving hit time, increasing bandwidth, dropping miss penalty, and reducing miss rate. Compiler optimization and prefetching techniques for main memory management techniques will be discussed in next section.

2.2.1 Small and Multi-level Cache

Cache is extremely expensive, and small cache decreases hit time but increases the miss rate [23]. Thus, many computers use multiple levels of caches to address the trade-off between cache latency and hit rate [47]. Small fast caches on chip are backed up by larger slower caches of separate memory chips. The multiple levels of caches to

use could yield an overall improvement in performance [38]. Some modern computers have begun to have three levels on chip cache, such as the AMD Phenom II has 6MB on chip L3 cache and Intel i7 has an 8MB on chip L3 cache [54].

2.2.2 Pipelined Cache

Cache could be pipelined for more stages to improve the clock frequency access, which gives fast clock cycle time and high bandwidth [24]. For example, MIPS R4000 divided the cache access into three independent stages [27]. Two pipeline stages are needed to access the on-chip cache and the third one is needed to perform the tag check; the instruction tag is checked simultaneously to decode. As a result, the latency of the cache lookup and tags comparison can be dramatically decreased by pipelined cache. [37] showed that a significant performance advantage is gained by using two or three pipeline stages to fetch data from the cache for high clock rate processors.

2.2.3 Trace Cache

A trace cache, proposed by [45], is a mechanism for increasing the instruction fetch bandwidth by storing of instructions in their dynamic order of execution rather than the static, sequential program order. Trace Cache is no longer having the problem of latency caused by generation of pointers to all of noncontiguous instruction blocks. In addition, trace cache is inherently a structure that uses repeating branch behavior and works well for programs with repetition structures. Trace cache has the benefits to deliver multiple blocks of instructions in the same cycle without support from a compiler and modifying instruction set, also there is no need to rotate, shift basic blocks to create dynamic instruction sequence. However, the disadvantage is that a trace cache stores a lot of redundant information [5]. Pentium 4 includes trace cache to store maximally 12K decoded micro-operations, and it uses virtual addresses so no need for address translation [55]. In [43], a software trace cache is used to improve in the instruction cache hit rate and instruction fetch bandwidth, the results showed it was effectively useful for not-taken branches and long chains of sequential instructions.

2.2.4 Nonblocking cache

A cache miss occurs when a memory request cannot find the address in cache, which results the process stalls to degrade the system performance. A non-blocking cache is employed to exploit post-miss operations by continuing to fetch during cache misses as long as dependency constraint are observed [3, 10]. [15] reported non-blocking cache provided a significant performance improvement, they simulated on the SPEC92 benchmarks, and the results reduced 21% of run-time with a 64-byte, 2-way set associative cache with 32 cycle fetch latency. The Intel Pentium Pro use this technology for their L2 cache, the processor could issue up to 4 cache requests at a time [4].

2.2.5 Multibanked caches

Caches are often partitioned into independent banks to run in parallel so as to increase cache bandwidth and to keep up with the speed of the CPU [44]. Memory usually is split by address into the cache banks. For example, AMD Athlon 64 has 8 banks, and each cycle it can fetch two 8-byte data from different banks. However, at the same time only one access is allowed to the same bank that is single-ported, such as MIPS R10000 Superscalar Microprocessor [60].

Since the mapping of addresses to banks affects the performance, the simplest mapping is called sequential interleaving to spread the block addresses sequentially across the banks. [12] also proposed vertical interleaving to divide memory banks in a cache into vertically arranged sub-banks. The result showed up to 48% reduction in maximum power density with two banks and up to 67% reduction with four banks on running Alpha binaries.

3 Memory Replacement Strategies

In a paged virtual memory system, when all of the page frames are occupied and a process references a nonresident page, the system should not only bring in a new memory page from auxiliary storage, but select first a victim page to be replaced by the incoming page. Pages swapped out of the main memory must be chosen carefully, otherwise it causes redundant memory reads and writes if a page is immediately requested again after it has been removed from memory, and system performance will degrade for a lot of the references of the secondary storage device.

The replacement algorithms can be divided into static page replacement algorithms, dynamic page replacement algorithms, and working set algorithms.

3.1 Static page replacement algorithms

Static page replacement algorithms all assume that each program is allocated a fixed amount of memory when it begins to execute, and does not request further memory during its lifetime.

3.1.1 Optimal algorithm (also called OPT or MIN)

It replaces the page that will not be referenced again until furthest in the future or has the longest number of page requests before it is referenced [42]. Use of this replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames. However, the OPT algorithm is impractical unless you have full prior knowledge of the reference stream or a record of past behavior that is incredibly consistent. As a result, the OPT algorithm is used mainly as a benchmark to which other algorithms can be compared.

3.1.2 First-In-First-Out algorithm (FIFO)

FIFO selects the page that has been in the system the longest (or first-in) to be removed. Conceptually, it works as

a limited size queue, with items being added at the tail. When the queue is full, the first page to enter is moved out of the head of the queue. Similar to RAND, FIFO blatantly ignores trends and might choose to remove a heavily referenced page. Although FIFO can be implemented with relatively low overhead using a queue, it is rarely used because it does not take advantage of locality trends.

A modification to FIFO that makes its operation much more useful is the second-chance and clock page replacement algorithms. The only modification here is that a single bit is used to identify whether or not a page has been referenced during its time in the FIFO queue. The second-chance algorithm examines the referenced bit of the oldest page; if this bit is off, it immediately selects that page for replacement. Otherwise, it turns off the bit and moves the page to the tail of the FIFO queue. Active pages will be kept at the tail of the list and thus remain in main memory. The clock page replacement works essentially the same as the second-chance algorithm, except arranging the pages in a circular list rather than a linear list [2]. Each time a page fault occurs, a list pointer moves around the circular list like the hand of a clock. When a page's reference bit is turned on, the pointer will turn the reference bit off and move to the next element of the list. The clock algorithm places the new arrival page in the first page it encounters with the referenced bit turned off.

3.1.3 Least-Recently-Used algorithm (LRU)

LRU relies on a process's recent past behavior as an indication of the near future behavior (temporal locality). It selects the page that has been the longest time in memory without being referenced when picking a victim. Although LRU can provide better performance than FIFO, the benefit comes at the cost of system overhead [57]. For example, one implementation of LRU algorithm - a stack algorithm - contains one entry for each occupied page frame. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom. When an existing page must be replaced to make room for an incoming one, the system replaces the entry at the bottom of the stack. The substantial overhead happens because the system must update the stack every time a page is referenced.

3.1.4 Least-Frequently-Used algorithm (LFU)

Often confused with LRU, least-frequently-used (LFU) algorithm selects a page for replacement if it has not been used often in the past. This strategy is based on the intuitively appealing heuristic that a page that is not being intensively referenced is not as likely to be referenced in the future. LFU can be implemented using a counter associated with each page, and the counter is updated each time its corresponding page is referenced; however, this can incur substantial overhead. The LFU algorithm could easily select incorrect pages for replacement. For instance, the least-frequently used page could be the page brought into main memory most recently. Furthermore, it tends to respond slowly to locality changes [22]. When a program either changes its set of active pages or terminates and is replaced by a different program, the frequency count will cause pages in the new locality to be immediately replaced since their frequency is at a much lower than the pages associated with the previous program. Because the context has changed and the pages swapped out will most likely be happening again soon (the principle of locality), a period of thrashing will likely occur.

3.2 Dynamic page replacement algorithms

When page allocation can change, static algorithms cannot adjust to keep the performance optimized. For example, a program rapidly switches between needing relatively large and relatively small page sets or localities [4]. Furthermore, it is incredibly difficult to find the optimal page allocation since a full analysis is rarely available to a virtual memory controller.

However, dynamic paging replacement algorithms could adjust and optimize available pages based on reoccurring trends. This controlling policy is also known as prefetch paging.

3.2.1 SEQ algorithm

Glass and Cao [21] proposes a new virtual memory page replacement algorithm, SEQ. Usually SEQ works like LRU replacement, however, it monitors page faults when they occur, and finds long sequences of page faults to contiguous virtual addresses. When such sequences are detected, SEQ performs a most-recently-used replacement that mimics OPT algorithm. For applications that have sequential behavior, SEQ has better performance when compared to LRU; for other types of applications, it performs the same as LRU. Their simulation results on Solaris 2.4 showed that for a large class of applications with clear access patterns, SEQ performs close to the optimal replacement algorithm, and it is significantly better than LRU. For other applications, SEQ algorithm performance is similar to LRU algorithm.

3.2.2 Adaptive prefetching algorithm

[19, 62] present a disk prefetching algorithm based on on-line measurements of disk transfer times and of inter-page fault rates to adjust the level of prefetching dynamically to optimize the system performance. In [19], the prefetching algorithm is proposed using life-time function for a single process execution model. Theoretical model is given and an analytical approach is extended to a multiprogramming environment using a queuing network model. Simulation experiments are provided to show the effective performance of the prefetching algorithm for some synthetic traces. [62] further studies this prefetching adaptive algorithm for the multiprocess execution models and uses trace-driven simulations to evaluate the effectiveness on real workloads. The adaptive prefetching algorithm could always keep the system efficiency and the system performance optimal.

3.2.3 Early eviction LRU (EELRU)

[52] introduces early eviction LRU (EELRU), which adjusts its speed of adaption on aggregating recent

information to recognize the reference behavior of a workload. The authors prove that EELRU offers strong theoretical guarantees of performance relative to the LRU algorithm: While it can be better than LRU by a large factor, on the other side, it can never be more than a factor of 3 worse than LRU.

Another adaptive dynamic cache replacement algorithm extended on EELRU is proposed based on prefix caching for a multimedia servers cache system [25]. Two LRU page lists for the prefix are stored in the cache; one maintains the prefix that has been requested only once recently, while the other maintains that has been requested at least twice recently. The simulation results show that the algorithm works better than LRU by maintaining the list of recently evicted videos.

3.2.4 ARC and CAR algorithms

ARC (Adaptive Replacement Cache) algorithm is developed by IBM Almaden Research Center in IBM DS6000/DS8000 storage controllers and works better than LRU [34]. It divides the cache directory into two lists, for storing recently and frequently referenced entries. In addition, a ghost list is used for each list to keep of the history of recently evicted cache entries. ARC's dynamic adaption exploits both the recency and the frequency features of the workload in a self-tuning way, and it provides a low-overhead alternative but outperforms LRU across a wide range of workloads and cache sizes.

CAR (Clock with Adaptive Replacement) algorithm inherits virtually all advantages of ARC [2], it uses two clocks with recency and frequency referenced entries. The sizes of the recently evicted pages are adaptively determined by using a precise history mechanism. Simulations demonstrate that CAR's performance is comparable to ARC, but substantially better than both LRU and CLOCK algorithms.

3.2.5 Working set algorithms

Denning's working set theory was developed to study what a favored subset is and how to maintain it in main memory to achieve the best performance [13]. It stated that for a program to run efficiently, the system must keep the pages of program's working set in main memory, otherwise excessive paging activity may happen to cause low processor utilization - thrashing. The working set is defined as W (t, w), where w is the process's working set window size referenced during the process time interval t − w to current process time interval t in Figure 1. A working set memory management policy tries to keep only the current working set pages in main memory to exploit the performance in the execution of that process. However, the policy is usually heuristic since it is so hard for the system to know the current working set pages in advance.

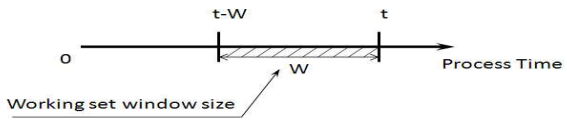


Figure1. Working set model

Localities may change; however, generally locality sets repeat within the global locality. Working set algorithms assume during the working set time-window, the program could add or remove pages until the time has expired [33]. When the window finishes, the time window may be dynamically adjusted to provide maximal correspondence with locality changes [13]. The adjustments can be made in a variety of ways, but mainly as a function of the rate of page faults occurring within the program.

Working set algorithms do not always use a specific time window to determine the active set. For example, various page fault frequency (PFF) algorithms can be used to monitor the rate at which a program incurs faults [48]. It is similar to modifying the time interval but is not subject to a minimal time for change to occur: When locality transits, page allocation or release may occur rapidly. PFF has its limitations on the some applications. [33] gave an example that requires unrelated references to a database, causing a large fault frequency. In this case, the program would not benefit from keeping the old references in memory. Rapid changes in the fault frequency due to this kind of access would result in either wasted page allocation or rapid thrashing with the algorithm, both detracting from its usefulness. Other working set methods like the working set clock algorithms (WSClock) [8] could closely approximate a static method on a global scale. Frames are considered for replacement by a pointer moving clockwise along a circular list. Although WSClock algorithms generally behave like LRU, their actual performance can differ based on timing parameters and selection criteria.

4 Memory Optimization Techniques

4.1 DRAM Chip Design

The main memory is generally made up of DRAM and has relatively large storage capacity as compared to a cache (SRAM), but it is slower, cheaper, and denser than a cache. Since the memory design needs to decrease the processor-memory gap, technologies usually are developed to have a greater bandwidth. The first innovation is a new revolutionary addressing scheme by multiplexed row and column addresses lines to cut the number of address pins in half, introduced by Robert Proebsting in Mostek MK4096 in 1975 [16]. This design sent row access strobe (RAS) first and then the column access strobe (CAS), to reduce the cost and space by fitting a 4K DRAM into a 16 pin package instead of previous 22 pin package.

Fast page mode is one improvement on conventional DRAM when the row- address is held constant and data from multiple columns is read without another row access time [36]. An additional basic change is Synchronous DRAM (SDRAM), unlike the conventional DRAMs are asynchronous to the memory controller, it adds a clock signal to the DRAM interface and has a register that holds a bytes-per-request value, and returns many bytes over several cycles per request [26]. Another major innovation to increase bandwidth is DDR (double data rate) SDRAM, which doubles the data bandwidth of the bus by reading and writing data on both the

rising and falling edges of the clock signal [41].

The above optimization techniques exploit the high potential DRAM bandwidth with adding little cost to the system, there are many more techniques we cannot list out since of the space limitation, such as Extended Data Out (EDO) DRAM, Enhanced Synchronous (ES) DRAM, Synchronous Link (SL) DRAM, Rambus DRAMs (RDRAM), Direct Rambus (DRDRAM), Fast Cycle (FC) DRAM, etc.

4.2 Software Optimization

Most of the techniques require changing the hardware; however, pure software approach such as compiler optimization could also improve the memory performance. Code can easily be rearranged to improve temporal or spatial locality to reduce misses. For example, interchanging independent statements might reduce instruction miss rates by reducing conflict misses [49]. Another example is called branch straightening, when the compiler predicts that a branch to happen, it rearranges program code by swapping the branch target block with the block sequentially right after the branch. Branch straightening could improve spatial locality.

Code and data rearrangement are mainly used on loop transformations. Loop inter-change is to exchange the order of the nested loop, making the code access the data in the order they are stored [58]. Loop fusion is a program transformation by fusing loops that access similar sets of memory locations. After fusion, the accesses are gathered together in the fused loop and can be reused easily. In [61], authors combined loop interchange and loop fusion together and applied on a collection of benchmarks, and the results indicate that their approach is highly effective and ensures better performance. When some arrays are accessed by rows and some by columns and both rows and columns are used in every iteration, Loop Blocking is used to create blocks instead of working on entire rows or columns to enhance reuse of local data [29].

4.3 Prefetching

Demand fetching can be minimized if we can successfully predict which information will be needed and fetch it in advance - prefetching. If the system is able to make correct predications about future page uses, the process's total runtime can be reduced [7, 53]. Prefetching strategies should be designed carefully. If a strategy requires significant resources or inaccurately preloads unneeded pages, it might result in worse performance than in a demand paging system. Assume that s pages are prefetched and a fraction α of these s pages is actually used ($0 \leq \alpha \leq 1$). The question is whether the cost of the $s \times \alpha$ saved page faults is greater or less than the cost of prepaging $s \times (1 - \alpha)$ unnecessary pages. If α is close to 0, prefetching loses; if α is close to 1, prefetching wins.

Prefetching strategies often exploit spatial locality. Furthermore, the address and size of the prefetched data are derived when considering the reference history, access pattern, and trajectory of prior memory accesses. Important criteria that determine the success of a prefetching strategy include:

- prepaged allocation - the amount of main memory allocated to prepaging
- the number of pages that are preloaded at once
- the algorithm used to determine which pages are preloaded.

Prefetching strategies are applied in different memory levels, such as caches [40], main memory [53], and disks [19 62]. Next we discuss the categories of hardware prefetching and software prefetching.

4.3.1 Hardware Prefetching

Both instructions and data can be prefetched. Instruction prefetch is happening normally outside the cache. Instructions following the one currently being executed are loaded into instruction stream buffer. If the requested instruction is already in the buffer, no need to fetch it again but request the next prefetch. Hardware data prefetching is used to exploit of run-time information without the need for programmer or compiler intervention. The simplest prefetching algorithm is sequential prefetching algorithm, such as one block lookahead (OBL) in HP PA7200 [9]. An adaptive sequential prefetching algorithm is proposed for a dynamic k value to match k with the degree of spatial locality of the program at a particular time. Several algorithms have been developed by using special cache called reference prediction table (RPT) to monitor the processor's referencing pattern to prefetching with arbitrary strides [1, 17]. In [10], authors found out the RPT may not be able to find an access pattern for indirect ad- dressing mode and the RPT only worked after an access pattern has been established. However, during steady loop state, the RPT could dynamically adjust its prefetching stride to achieve a good performance.

4.3.2 Software Prefetching

Even there are some software prefetching on instructions [18], most software prefetching algorithms are working for data only, applying mostly within loops for large array calculations for both hand-coded and automated by a compiler [30]. [35] applied prefetching for affine array references in scientific programs, locality analysis is conducted to find the part of array references suffered from cache misses. Then loop unrolling and loop peeling to extract the cache-missing memory references are used, finally to apply prefetch instructions on the isolated cache-missing references. Jumper pointer prefetching connects non-consecutive link elements by adding additional pointers into a dynamic data structure [46]. It is effective for limited work available between successive dependent accesses. Jumper pointer prefetching can be extended to prefetch array [28], which overcomes the problem that jumper pointer prefetching is unable to prefetch the early nodes because of no pointers. To have prefetch instructions embedded in program incurring overhead, so compilers must ensure that benefits exceed the overheads.

5 Conclusions

In this survey, we review the memory system design, memory management techniques, and optimization

approaches. Given the constraints on the length of this paper, we have not addressed the role of memory management and optimization techniques in embedded system such as handheld devices. [31] has examined data and memory optimization techniques to memory structures at different levels for embedded systems. And a survey has been conducted for techniques for optimizing memory behavior of embedded software in [59].

Memory system will remain the vital part of the computers unless the conventional Von Neumann model is outdated. DRAM has been dominating the primary memory since 1971. The processor-memory latency gap continues to grow because the continual improvements in processor cycle speed is much faster than improvements in DRAM access latency. Many processors and memory designs, architectures, optimization techniques are developed and proposed to minimize the gap. However, there is no surprise that another technology will replace DRAM in the price/performance consideration soon. For example, Magnetic RAM (MRAM) has the advantage of being non-volatile. Another choice is optical storage, which could transport the entire system memory image in a small three dimensional carrier. No matter what technology is, it should balance cost, performance, power consumption, and complexity.

6 References

[1] Baer, J.L. and Chen, T.F. (1991) An Effective On-chip Preloading Scheme to Reduce Data Access Penalty, Proceeding Supercomputing 91, Albuquerqu, NM, November, 176-186.

[2] Bansal, S. and Modha, D.S. (2004) CAR: Clock with Adaptive Replacement, Proceeding of USENIX Conference on File and Storage Technologies, 187-200, CA.

[3] Belayneh, S. and Kaeli, D. (1996) A Discussion on Non-blocking/lockup-free Caches, ACM SIGARCH Computer Architecture News, 24(3), 18-25.

[4] Bhandarkar, D.; Ding, J. (1997) Performance Characterization of the Pentium Pro Processor, Proceeding of High-Performance Computer Architecture, 1-5 Feb, 288- 297, San Antonio, Texas, USA.

[5] Black, B., Rychlik, B., and Shen, J.P. (1999) The Block-based Trace Cache, Proceeding of the 26th Annual International Symposium on Computer Architecture,196-207.

[6] Burger, D., Goodman, J.R. and Sohi, G.S. (1997) Memory Systems. The Computer Science and Engineering Handbook, 47-461.

[7] Cao, P., Felten, E.W., Karlin, A.R. and Li, K. (1995) A Study of Integrated Prefetching and Caching Strategies, Proc. of ACM SIGMETRICS, 188-197.

[8] Carr, W.R., Hennessy, J.L. (1981) WSClock-A Simple and Effective Algorithm for Virtual Memory Management, Proc. Of the ACM SOSP, 87-95.

[9] Chan, K.K., et al. (1996) Design of the HP PA 7200 CPU, Hewlett-Packard Journal, 47(1), 25-33.

[10] Chen, T.; Baer, J. (1992) Reducing Memory Latency via Non-blocking and Prefetching Caches, Proceeding of ACM SIGPLAN Notice, 27(9), 51-61.

[11] Chen, T.; Baer, J. (1994) A Performance Study of Software and Hardware Data Prefetching Schemes, Proceeding of the 21st Annual International Symposium on Computer Architecture, Chicago, IL, April, 223-232.

[12] Cho, S. (2007) I-Cache Multi Banking and Vertical Interleaving, Proceeding of ACM Great Lakes Symposium on VLSI, March 11-13, 14-19, Stresa-Lago Maggiore, Italy.

[13] Denning, P.J. (1980) Working Sets Past and Present, IEEE Transactions on Software Engineering, SE6(1), 64-84.

[14] Denning, P.J. (2005) The Locality Principle, Communications of the ACM, 48(7), 19 - 24.

[15] Farkas, K.I.; Jouppi, N.P.; Chow, P. (1994) How Useful Are Non-Blocking Loads, Stream Buffers, and Speculative Execution in Multiple issue Processors? Western Research Laboratory Research Report 94/8.

[16] Foss, R.C. (2008) DRAM - A Personal View, IEEE Solid-State Circuits Newsletter 13(1), 50-56.

[17] Fu, J.W.C, Patel, J.H., and Janssens, B.L. (1992) Stride Directed Prefetching in Scalar Processors, Proceeding of 25th International Symposium on Microarchitecture, Portland, OR, December, 102-110.

[18] Galazin, A.B., Stupachenko, E.V., and Shlykov, S.L. (2008) A Software Instruction Prefetching Method in Architectures with Static Scheduing, Programming and Com- puter Software, 34(1), 49-53.

[19] Gelenbe, E. and Zhu, Q. (2001) Adaptive Control of Prefetching, Performance Evaluation, 46, 177-192.

[20] Ghasemzadeh, H., Mazrouee,S., Moghaddam, H.G., Shojaei, H. and Kakoee, M.R. (2006) Hardware Implementation of Stack-Based Replacement Algorithms, World Academy of Science, Engineering and Technology, 16, 135-139.

[21] Glass, G. and Cao, P. (1997) Adaptive page replacement based on memory reference behavior, Proc of the 1997 ACM SIGMETRICS, 25(1), 115-126.

[22] Gupta, R.K. and Franklin, M.a. (1978) Working Set and Page Fault Frequency Re- placement Algorithms: A Performance Comparison, IEEE Transactions on Computers, C-27, 706-712.

[23] Handy, J (1997) The Cache Memory Book, Morgan Kaufmann Publishers.

[24] Hennessy, J.L. and Patterson, D.A. (2007) Computer architecture: a quantitative approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, 4th Edition.

[25] Jayarekha, P.; Nair, T.R (2009) Proceeding of InterJRI Computer Science and Networking, 1(1), Dec, 24-30.

[26] Jones, F. et al., (1992) A New Era of Fast Dynamic RAMs, IEEE Spectrum, 43-49. [56] Kaplan, K.R. and Winder, R.O. (1973) Cache-based Computer Systems, IEEE Computer, 6(3), 30-36.

[27] Kane, G. and Heinrich J. (1992) MIPS RISC Architecture, Prentice Hall.

[28] Karlsson, M., Dahlgren, F., and Stenstrom, P. (2000) A Prefetching Technique for Irregular Accesses to Linked

Data Structures, Proceeding of the 6th International conference on High Performance Computer Architecture, Toulouse, France, January, 206-217.
[29] Lam, M.; Rothberg, E.; Wolf, M.E. (1991) The Cache Performance and Optimization of Blocked Algorithms, Proceeding of the Fourth International Conference on ASPLOSIV, Santa Clara, Apr, 63-74.
[30] Luk, C.K and Mowry, T.C (1996) Compiler-based Prefetching for Recursive Data Structures, Proceeding of 7th Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, October, 222-233.
[31] Mahapatra, N.R., and Venkatrao, B. (1999) The processor-memory bottleneck: problems and solutions, Crossroads, 5(3).
[32] Mahajan, A.R. and Ali, M.S. (2007) Optimization of Memory System in Real-time Embedded Systems, Proceeding of the 11th WSEAS International Conference on Computers. 13-19.
[33] Marshall, W.T. and Nute, C.T. (1979) Analytic modeling of working set like replacement algorithms, Proc. of ACM SIGMETRICS, 65-72.
[34] Megiddo, N.; and Modha, D.S. (2003) ARC:A Self-tuning, Low Overhead Replacement Cache, Proceeding of 2nd USENIX conference on File and Storage Technologies, 115-130, San Franciso, CA.
[35] Mowry, T. (1991) Tolerating Latency through Software-controller Prefetching in Shared-memory Multiprocessors, Journal of Parallel and Distributed Computing, 12(2), 87-106.
[36] Ng, Ray (1992) Fast Computer Memories, IEEE Spectrum, 36-39.
[37] Olukotun, K., Mudge, T., and Brown, R. (1997) Multilevel Optimization of Piplelined Caches. IEEE Transactions on Computers, 46, 10, 1093-1102.
[38] Ou, L., He, X.B., Kosa, M.J., and Scott, S.L. (2005) A Unified Multiple-Level Cache for High Performance Storage Systems, mascots, 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 143-152.
[39] Panda, P.R. et al (2001) Data and Memory Optimization Techniques for Embedded System, ACM Transactions on Design Automation of Electronic Systems, 6(2), 149-206.
[40] Perkins, D.R. (1980) The Design and Management of Predictive Caches, PH.D dissertation, UC., San Diego.
[41] Peters, M. (2000) Enhanced Memory Systems, Personal Communications, September.
[42] Prieve, B.G. and Fabry, R.S. (1976) VMIN - An Optimal Variable Space Page Re- placement Algorithm, Communications of the ACM, 19(5), 295-297.
[43] Ramirez, A., Larriba-Pey, J.L., and Valero, M. (2005) Software Trace Cache, IEEE Transactions on Computer, 54(1), 22-35.
[44] Rivers, J.A.; Tyson, G.S.; Dividson, E.S.; Austin, T.M. (1997) On High-Bandwidth Data Cache Design for Multi-Issue Processors, Proceeding of International Symposium of Microarchitecture, Dec, 46-56.
[45] Rotenerg, E., Bennett, S., and Smith, J.E. (1999) A Trace Cache Microarchitecture and Evaluation, IEEE Transactions on Computers, 48(2), 111-120.
[46] Roth, A. and Sohi, G., (1999) Effective Jump-pointer Prefetching for Linked Data Structures. Proceeding of the 26th International Symposium on Computer Architecture, Atlanta, GA, May, 111-121.
[47] Pas, R. (2002) Memory Hierarchy in Cache-Based Systems, Sun Microsystems.
[48] Sadeh, E. (1975) An Analysis of the Performance of the Page Fault Frequency (PFF) Replacement Algorithm, Proc of the fifth ACM SOSP, 6-13.
[49] Samples, A.D.; Hilfinger, P.N. (1988) Code Reorganization for Instruction Caches, University of California at Berkeley, Berkeley, CA.
[50] Serhan, S.I. and Abdel-Haq, H.M. (2007) Improving Cache Memory Utilization, World Academy of Science and Technology, 26, 299-304.
[51] Shemer, J.E. and Shippey, B. (1966) Statistical Analysis of Paged and Segmented Computer Systems. IEEE Trans. EC-15, 6, 855-863.
[52] Smaragdakis, Y., Kaplan, S. and Wilson, P. (2003) The EELRU Adaptive Replace- ment Algorithm, Performance Evaluation, 53(2), 93-123.
[53] Smith, A.J. (1978) Sequential Program Prefetching in Memory Hierarchies, IEEE Computer, 11(12), 7-21.
[54] Sondag, T. and Rajan, H. (2009) A Theory of Reads and Writes for Multi-level Caches. Technical Report 09-20a, Computer Science, Iowa State University.
[55] Tuck, N; Tullsen, D.M. (2003) Initial Observations of the Simultaneous Multithread- ing Pentium 4 Processor, Proceeding of Parallel Architectures and Compilation Techniques, Oct, 26-34.
[56] A.M Turing (1937) On Computable Numbers, with an Application to the Entscheidungsproblem, Proceedings of the London Mathematical Society, 42(2), 230-265.
[57] Turner, R. and Levy, H. (1981) Segmented FIFO Page Replacement, Proc. of ACM SIGMETRICS on Measurement and Modeling of Computer System, 48-51.
[58] Wolf, M.E., Lam, M. (1991) A Data Locality Optimizing Algorithm. Proceeding of the SIGPLAN conference on Programming Language Design and Implementation, Toronto, 26(6), 30-44.
[59] Wolf, W, Kandemir, M. (2003) Memory System Optimization of Embedded Soft- ware. Proceeding of IEEE, Jan, 91(1), 165-182.
[60] Yeager, K.C.(1996) The MIPS R10000 Superscalar Microprocessor, IEEE Micro, Apr., 16(2), 28-40, 1996.
[61] Yi, Q. and Kennedy, K. (2004) Improving Memory HierarchyPerformance through Combined Loop Interchange and Multi-Level Fusion, International Journal of High Performance Computing Applications, 18(2), 237-253.
[62] Zhu, Q., Gelenbe, E. and Qiao, Y (2008) Adaptive Prefetching Algorithm in Disk Controllers, Performance Evaluation, 65(5), 382-395.

Synchronization of a complex dynamical network with nonidentical nodes via dynamic feedback control

T.H. Lee[1], J.H. Park[1], H.Y. Jung[1], S.M. Lee[2]
[1]Nonlinear Dynamics Group/Dept. EE/ICE, Yeungnam University, Kyongsan, Republic of Korea.
[2]Department of Electronic Engineering, Daegu University, Gyungsan, Republic of Korea.

Abstract— *This paper considers synchronization problem of a complex dynamical network with nonidentical nodes. For this problem, a dynamic feedback controller is designed to achieve the synchronization of the network. Based on Lyapunov stability theory and linear matrix inequality framework, the existence condition for feasible controllers is derived in terms of linear matrix inequalities. The condition can be solved easily by the application of convex optimization algorithms. Finally, the proposed method is applied to a numerical example in order to show the effectiveness of our result.*

Keywords: Complex dynamical network, synchronization, nonidentical node, dynamic feedback control.

1. Introduction

During the last decade, complex dynamical networks, which are a set of interconnected nodes with specific dynamics, have been attracted increasing attention in various fields such as physics, biology, chemistry and computer science [1]. As science and society develop, our everyday lives have been closed to complex networks, for instance, transportation networks, World Wide Web, coupled biological and chemical engineering systems, neural networks, social networks, electrical power grids and global economic markets. Many of these networks exhibit complexity in the overall topological and dynamical properties of the network nodes and the coupled units. Recently, one of the significant and interesting phenomena in complex dynamical network is the synchronization. Synchronization of complex dynamical networks can be divided into two points of view. One is the synchronization of a complex network that is called 'inner synchronization' [2]-[5]. It means that all the nodes in a complex network eventually approach to trajectory of a target node. Another is called 'outer synchronization' [6]-[8] which considers the synchronization between two or more complex networks. In this paper, a new control problem for inner synchronization will be investigated.

The random-graph model had become a basics of modern network theory since proposed by Erdös and Renyi [9]-[10]. In a random network, each pair of nodes is connected with a certain probability. Watts and Strogatz [11] introduced useful network model to translate from a regular network to a random network, it is called small-world network. Then, Newman and Watts [12] modified it to generate another variant of the small-world model. And then, Barabasi and Albert [13] proposed a scale-free network model, in which the degree distribution of the nodes follows a power-law form . Thereafter, small-world and scale-free networks have been extensively investigated.

Synchronization of a complex dynamical network have been well noticed that many researchers adopt the assumption that all nodes dynamics are identical [2]-[3]. However, this assumption about identical nodes is unlikely environment in most of complex dynamical networks. For example, in a swarm robot system, every individual robots have different dynamics of them, and even if the swarm robot system is consisted of same robots, it has possibility to be nonidentical network system due to uncertainties, saturation and so on. When the nodes of a complex network is nonidentical, they will show different dynamics. It should be noted that the synchronization schemes for networks with identical nodes is good for nothing. Therefore, the further investigation of new synchronization schemes for a complex dynamical network with nonidentical nodes is necessary. In this regard, only a few papers have been reported until now [4]-[5].

In this paper, we will investigate the synchronization of a complex network with nonidentical node via dynamic feedback control unlike previous works which usually treated a complex network with identical nodes. Until now, in order to treat this kind of problem for a complex network, several control schemes such as adaptive control ([2], [7]-[8]) and pinning control ([2]-[3], [6]) are applied. However, to the best of the authors' knowledge, the synchronization problem via dynamic feedback controller for complex networks has not been investigated up to now. In some real control situations, there is a strong need to construct a dynamic feedback controller instead of a static feedback controller in order to obtain a better performance and dynamical behavior of the state response. The dynamic controller will provide more flexibility compared to the static controller and the apparent advantage of this type of controller is that it provides more free parameters for selection [14]. So it is very worth to consider the design problem of dynamic controller for synchronization in a complex network. The existence condition of such controller is derived in terms of linear matrix inequalities (LMIs) which can be easily solved by standard convex optimization algorithms [15].

Notation: $X > 0$ (respectively, $X \geq 0$) means that the matrix X is a real symmetric positive definite matrix (respectively, positive semi-definite). I_n denotes the n-dimensional identity matrix. $\otimes$ denotes the notation of Kronecker product.

2. Problem statement and preliminaries

Consider a delayed complex dynamical network consisting of N linearly coupled nonidentical nodes described by

$$\dot{x}_i(t) = f_i(x_i(t)) + \sum_{j=1}^{N} c_{ij}x_j(t) + u_i(t),\ i = 1, \ldots, N \quad (1)$$

where $x_i = (x_{i1}, x_{i2}, \ldots, x_{in})^T \in \mathbb{R}^n$ is the state vector of the ith node, $f_i : \mathbb{R}^n \to \mathbb{R}^n$ is a smooth nonlinear vector field, $u_i(t)$ is the control input of ith node, and c_{ij} is the coupling configuration parameter representing the coupling strength and the topological structure of the network, in which c_{ij} is nonzero if there is a connection from node i to node $j (i \neq j)$, and is zero otherwise. For simplicity, let us define $C = (c_{ij})_{N\times N}$. Also, the diagonal elements of matrix C are assumed that

$$c_{ii} = -\sum_{j=1, j\neq i}^{N} c_{ij},\ \ i = 1, \ldots, N. \quad (2)$$

Definition 1. A complex network is said to achieve asymptotical inner synchronization, if

$$x_1(t) = x_2(t) = \cdots = x_N(t) = s(t) \quad \text{as } t \to \infty,$$

where $s(t) \in \mathbb{R}^n$ is a solution of a target node, satisfying $\dot{s}(t) = f_s(s(t))$.

Here, define error vectors as follows :

$$e_i(t) = s(t) - x_i(t). \quad (3)$$

From Eq. (3), the error dynamics is given to

$$\begin{aligned} \dot{e}_i(t) &= f_s(s(t)) - f_i(x_i(t)) - \sum_{j=1}^{N} c_{ij}e_j(t) - u_i(t) \\ &= \bar{f}_i(e_i(t)) - \sum_{j=i}^{N} c_{ij}e_j(t) - u_i(t), \end{aligned} \quad (4)$$

where $\bar{f}_i(e_i(t)) = f_s(s(t)) - f_i(x_i(t))$.

Also, a vector-matrix form of Eq. (4) is described by

$$\dot{e}(t) = F(t) - C \otimes I_n e(t) - U(t) \quad (5)$$

where $F = \left[\bar{f}_1^T(e_1(t)), \bar{f}_2^T(e_2(t)), \ldots, \bar{f}_N^T(e_N(t))\right]^T$, $e = \left[e_1^T(t), \ldots, e_N^T(t)\right]^T$, and $U = \left[u_1^T(t), u_2^T, \ldots, u_N^T(t)\right]$.

3. Controller design

In this section, a dynamic feedback controller will be designed to achieve the synchronization goal.
In order to stabilize the error system given in Eq. (5), let's consider the following dynamic feedback controllers:

$$\begin{aligned} &\dot{\zeta}(t) = A_c \otimes I_n\zeta(t) + B_c \otimes I_n e(t), \\ &U(t) = C_c \otimes I_n\zeta(t) + F(t),\ \ \zeta(0) = 0, \end{aligned} \quad (6)$$

where $\zeta(t) \in \mathbb{R}^{Nn}$ is the controller state, and A_c, B_c and C_c are constant gain matrices of $N \times N$ dimensions.

Applying this controller (6) to error system (5) results in the following closed-loop system

$$\dot{z}(t) = H \otimes I_n z(t), \quad (7)$$

where

$$z(t) = \begin{bmatrix} e(t) \\ \zeta(t) \end{bmatrix} \in \mathbb{R}^{2Nn},\ \ H = \begin{bmatrix} -C & -C_c \\ B_c & A_c \end{bmatrix} \in \mathbb{R}^{N\times N}.$$

Then we have following main result.

Theorem 1. There exists a dynamic feedback controller given in Eq. (6) for synchronization of the complex network Eq. (1) if there exist positive-definite matrices $S, Y \in \mathbb{R}^{N\times N}$ and matrices $X_1, X_2, X_3 \in \mathbb{R}^{N\times N}$ satisfying the following LMIs :

$$\begin{bmatrix} -YC^T - CY - X_1 - X_1^T & -C + X_3 \\ \star & (2,2) \end{bmatrix} < 0 \quad (8)$$

and

$$\begin{bmatrix} Y & I_N \\ I_N & S \end{bmatrix} > 0. \quad (9)$$

where $(2,2) = -C^TS - S^TC + X_2 + X_2^T$.

Proof. Consider the following Lyapunov function: $V(t) = z^T(t)P \otimes I_n z(t)$ where $P \in \mathbb{R}^{2N\times 2N} > 0$. Then, the time derivative of the Lyapunov function is

$$\dot{V}(t) = z^T(t)(H^TP + PH) \otimes I_n z(t). \quad (10)$$

Here, let us define

$$\Sigma \equiv H^TP + PH. \quad (11)$$

Thus, if the inequality $\Sigma < 0$ holds, then it can be said that synchronization of a complex network with nonidentical node is achieved by our proposed dynamic controller. However, in the matrix Σ, the matrix $P > 0$ and the controller parameters A_c, B_c and C_c, which included in the matrix H, are unknown and occur in nonlinear fashion. Hence, the inequality $\Sigma < 0$ cannot be considered as an linear matrix inequality problem. In the following, we will use a method of changing variables such that the inequality can be solved as convex optimization algorithm [16].

First, partition the matrix P and its inverse as

$$P = \begin{bmatrix} S & J \\ J^T & T \end{bmatrix},\ \ P^{-1} = \begin{bmatrix} Y & M \\ M^T & W \end{bmatrix}, \quad (12)$$

where S, Y are positive-definite matrices, and $M, N \in \mathbb{R}^{N\times N}$ are invertible matrices. It should be pointed out that the equality $P^{-1}P = I_{2N}$ gives that

$$MJ^T = I_N - YS. \tag{13}$$

Define two matrices as

$$F_1 = \begin{bmatrix} Y & I_N \\ J^T & 0 \end{bmatrix}, \quad F_2 = \begin{bmatrix} I_N & S \\ 0 & J^T \end{bmatrix}. \tag{14}$$

Then, it follows that

$$PF_1 = F_2, \quad F_1^T P F_1 = F_1^T F_2 = \begin{bmatrix} Y & I_N \\ I_N & S \end{bmatrix} > 0. \tag{15}$$

Now, postmultiplying and premultiplying the matrix inequality, $\Sigma < 0$, by the matrix F_1^T and by its transpose, respectively, gives

$$F_2^T H F_1 + F_1^T H^T F_2 < 0. \tag{16}$$

By utilizing the relation Eqs. (12)-(15), it can be easily obtained that the inequality Eq. (16) is equivalent to

$$\begin{bmatrix} \Gamma_1 & \Gamma_2 \\ \star & \Gamma_3 \end{bmatrix} < 0 \tag{17}$$

where $\Gamma_1 = -YC^T - CY - MC_c^T - C_c M^T$, $\Gamma_2 = -C - YC^T S + YX_2^T - X_1 S + MA_c^T J^T$, $\Gamma_3 = -C^T S - S^T C + JB_c + B_c^T J^T$. By defining a new set of variables as follows: $X_1 = MC_c^T$, $X_2 = JB_c$, $X_3 = -YC^T S + YX_2^T - X_1 S + MA_c^T J^T$, then, Eq. (17) is simplified to LMI (8). And the LMI (9) is equivalent to the positiveness of P. This completes the proof. ■

Remark 1. Given any solution of the LMIs given in Eqs. (8) and (9) in Theorem 1, a corresponding controller of the form Eq. (6) will be constructed as follows:

- Compute the invertible matrices M and J satisfying Eq. (13) using matrix algebra.
- Utilizing the matrices M and J obtained above, solve the equations X_i for B_c, C_c and A_c (in this order).

4. Numerical example

In order to show the effectiveness of the proposed method, we present a numerical example which is inner synchronization of a complex network with five nonidentical nodes. Each nodes are different chaotic systems such as well-known Lorenz, Chen, Lü, Chen-Lee and Genesio-Tesi systems. They are typical benchmark three dimensional chaotic systems and their chaotic behavior are displayed in Fig. 1. Thus, the complex network system consisting of five nodes is described by:

$$\dot{x}_i(t) = f_i(x_i(t)) + \sum_{j=1}^{N} c_{ij} x_j(t) + u_i(t), \tag{18}$$

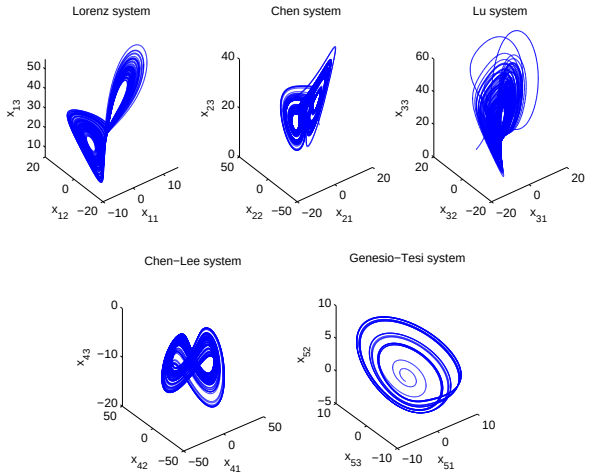


Fig. 1: Original chaotic behavior of each nodes

where $i = 1, \ldots, 5$ and

$$f_1(x_1(t)) = \begin{bmatrix} p_1(x_{12} - x_{11}) \\ p_3 x_{11} - x_{12} - x_{11}x_{13} \\ x_{11}x_{12} - p_2 x_{13} \end{bmatrix}$$

$$f_2(x_2(t)) = \begin{bmatrix} p_4(x_{22} - x_{21}) \\ (p_6 - p_4)x_{21} + p_6 x_{21} - x_{21}x_{23} \\ x_{21}x_{21} - p_5 x_{23} \end{bmatrix}$$

$$f_3(x_3(t)) = \begin{bmatrix} p_7(x_{32} - x_{31}) \\ p_9 x_{32} - x_{31}x_{33} \\ x_{31}x_{32} - p_8 x_{33} \end{bmatrix}$$

$$f_4(x_4(t)) = \begin{bmatrix} q_1 x_{41} - x_{42}x_{43} \\ -q_2 x_{42} + x_{41}x_{43} \\ -q_3 x_{43} + (1/3)x_{41}x_{42} \end{bmatrix}$$

$$f_5(x_5(t)) = \begin{bmatrix} x_{52} \\ x_{53} \\ -q_4 x_{51} - q_5 x_{52} - q_6 x_{53} + x_{51}^2, \end{bmatrix}$$

with the parameters $p_1 = 10$, $p_2 = 8/3$, $p_3 = 28$, $p_4 = 35$, $p_5 = 3$, $p_6 = 28$, $p_7 = 36$, $p_8 = 3$, $p_9 = 20$, $q_1 = 5$, $q_2 = 10$, $q_3 = 3.8$, $q_4 = 6$, $q_5 = 2.92$, and $q_6 = 1.2$.
Each nodes represent chaotic behavior as follows : $f_1(x_1(t))-$ Lorenz System, $f_2(x_2(t))-$ Chen System, $f_3(x_3(t))-$ Lü System, $f_4(x_4(t))-$ Chen-Lee System, $f_5(x_5(t))-$ Genesio-Tesi System. And a target node is also selected same one as first node, Lorenz system. In this example, random function, $|d(t)| < 10$, is used to every initial conditions of $x_i(0) = (d(t), d(t), d(t))$, $s(0) = (d(t), d(t), d(t))$, $(i = 1, 2, \ldots, 5)$ and coupling matrix, C, is given by

$$C = 0.2 \times \begin{bmatrix} -3 & 1 & 1 & 0 & 1 \\ 1 & -4 & 1 & 1 & 1 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 1 & 1 & -3 & 1 \\ 1 & 0 & 0 & 1 & -2 \end{bmatrix}. \tag{19}$$

In order to show original behavior of the complex network (18) with nonidentical node, the trajectories of the complex network (18) without controller is depicted in Fig 2.

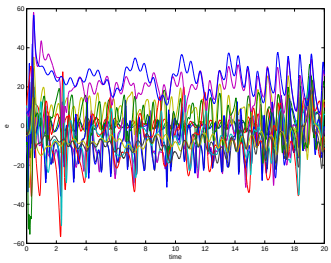


Fig. 2: Error signals of Example without control input

Now, we design a suitable dynamic feedback controller of the form Eq. (6) for system Eq. (18), which guarantees the asymptotic stability of the closed-loop system. By application of Theorem 1 to the system Eq. (18) and checking the feasibility of the LMIs given in Eqs (8) and (9), we can find that the LMIs are feasible by use of LMI control toolbox and obtain a possible set of solution of the LMIs. But, due to limitation of space, the solutions are omitted here. Then, by further calculation in light of Remark 1, we have a possible stabilizing dynamic feedback controller for the system Eq. (18):

$$A_c = \begin{bmatrix} -1.6000 & 0.2667 & 0.0376 & 0.0580 & -0.0638 \\ 0.3141 & -1.6010 & 0.0853 & 0.3113 & 0.1537 \\ 0.0212 & 0.2735 & -1.6185 & 0.0896 & -0.1972 \\ 0.0639 & 0.2963 & 0.0465 & -1.4779 & 0.1795 \\ 0.1297 & 0.1701 & -0.1499 & 0.1795 & -1.5026 \end{bmatrix}$$

$$B_c = \begin{bmatrix} -1.1000 & 0.2000 & 0.1000 & 0.0000 & 0.2000 \\ 0.2904 & -0.8863 & -0.1097 & 0.5507 & -0.5120 \\ 0.0294 & 0.8606 & -0.0314 & -0.0612 & -0.7567 \\ 0.0609 & -0.0716 & 0.2112 & -1.0067 & 0.1488 \\ 0.0330 & 0.4988 & -0.9123 & 0.0974 & 0.0563 \end{bmatrix}$$

$$C_c = \begin{bmatrix} -1.1000 & 0.2904 & 0.0294 & 0.0609 & 0.0330 \\ 0.2000 & -0.8863 & 0.8606 & -0.0716 & 0.4988 \\ 0.1000 & -0.1097 & -0.0314 & 0.2112 & -0.9123 \\ 0.0000 & 0.5507 & -0.0612 & -1.0067 & 0.0974 \\ 0.2000 & -0.5120 & -0.7567 & 0.1488 & 0.0563 \end{bmatrix}$$

The simulation result with control input is presented in Fig. 3. As seen in Fig. 3, the trajectories of error systems approach to zero as expected. This achieves asymptotic synchronization of the complex network (18).

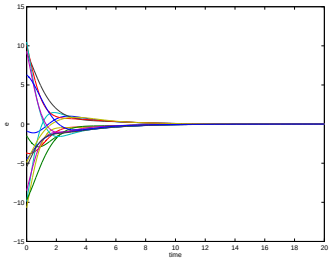


Fig. 3: Error signals of Example

5. Conclusions

In this paper, the asymptotic inner synchronization of a complex dynamical network has been studied. The nonidentical node was considered in the sense of reality. Unlike other works, a dynamic feedback controller was designed for the our synchronization scheme based on the Lyapunov method. Then, a criterion expressed by LMIs for stability of error dynamics was derived. Finally, one numerical example was illustrated to show the effectiveness of the designed controller.

Acknowledgements

This work was supported in part by MEST & DGIST (12-BD-0101, Renewable energy and intelligent robot convergence technology development.) This research was also supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2010-0009373).

References

[1] S.H. Strogatz. Exploring complex networks. *Nature 410 (2001) 268-276.*

[2] J. Zhou, J.A. Lu and J. Lu. Pinning adaptive synchronization of a general complex dynamical network. *Automatica 44 (2008) 996-1003.*

[3] L. Xiang and J.J.H. Zhu. On pinning synchronization of general coupled networks. *Nonlinear Dynamics 64 (2011) 339-348.*

[4] G. Solis-Perales, E. Ruiz-Velazquez, D. Valle-Rodriguez. Synchronization in complex networks with distinct chaotic noeds. *Commun. Nonlinear Sci. Numer. Simulat. 14 (2009) 2528-2535.*

[5] Q. Song, J. Cao, F. Liu. Synchronization of complex dynamical networks with nonidentical nodes. *Physics Letters A 374 (2010) 544-551.*

[6] D. Xu and Z. Su. Synchronization criterions and pinning control of general complex networks with time delay. *Appl. Math. Comput. 215 (2009) 1593-1608.*

[7] H. Tang, L. Chen, J. Lu and C.K. Tse. Adaptive synchronization between two complex networks with nonidentical topological structures. *Physica A 387 (2008) 5623-5630.*

[8] S. Zheng, Q. Bi and G. Cai. Adaptive projective synchronization in complex networks with time-varying coupling delay. *Physics Letters A 373 (2009) 1553-1559.*

[9] P. Erdös and A. Rényi. On random graphs. *Pub. Math. 6 (1959) 290-297.*

[10] P. Erdös and A. Rényi. On the evolution of random graphs. *publications Mathematiques. Institut de Hungarian Academy of Sciences, 5 (1960) 17-61.*

[11] D.J. Watts and S.H. Strogatz. Collective dynamics of small-world networks. *Nature 393 (1998) 440-442.*

[12] M.E.J. Newman and D.J. Watts. Renormalization group analysis of the small-world network model. *Physics Letters A 263 (1999) 341-346.*

[13] A.L. Barabási and R. Albert. Emergence of scaling in random networks. *Science 286 (1999) 509-512.*

[14] J.H. Park. Convex optimization approach to dynamic output feedback control for delay differential systems of neutral type. *J Optimization Theory Appl 127 (2005) 411-423.*

[15] B. Boyd, L.E. Ghaoui, E. Feron and V. Balakrishnan. Linear matrix inequalities in systems and control theory. *Philadelphia: SIAM, 1994.*

[16] C. Scherer, P. Gahinet, M. Chilali. Multiobjective output-feedback control via LMI optimization. *IEEE Transactions on Automatic Control 42 (1997) 896-911.*

SESSION

HPC AND MULTI-PROCESSOR MULTI-CORE SYSTEMS + DESIGN ISSUES + FPGA + GPU + NOC + EMBEDDED SYSTEMS

Chair(s)

TBA

A Novel Branch Predictor Using Local History for Miss-Prediction Bias

Lin Meng[1], **Katsuhiro Yamazaki**[1], **and Shigeru Oyanagi**[2]
[1]College of Science and Engineering, Ritsumeikan University, Kusatsu, Shiga, Japan
[2]College of Information Science and Engineering, Ritsumeikan University, Kusatsu, Shiga, Japan

Abstract— *Increasing accuracy of branch prediction is important for enhancing the performance of current processors. This paper discusses an interesting behavior of a current branch predictor that a large rate of miss-predictions is occupied by a few branches. We propose a novel branch prediction mechanism by using local history for miss-prediction biased branches. This mechanism is attached to a conventional branch predictor, and utilizes local history of biased branch instructions without conflict aliasing. Experiments are done by attaching proposed mechanism to several predictors. The results show that the proposed mechanism reduces the miss-predictions about 10%, and increases performance about 2% comparing to the conventional predictors at SPECint2000.*

Keywords: branch prediction; miss-prediction; miss-prediction bias;

1. Introduction

Current processors use deeper pipeline and wider instruction issue width for exploiting instruction level parallelism. However when a branch miss-prediction happens, it causes heavy miss-prediction penalty, such as wasting large number of cycles and power for miss-prediction recovery [1]. Hence, increasing accuracy of branch prediction is more important for improving the superscalar processor performance.

In 1981, the first branch predictor named Bimodal predictor [2] was proposed. Since then, many branch predictors have been proposed for increasing accuracy of branch predictions. Bimodal predictor used a 2 bits saturating counter to keep the behavior of branch. These 2 bits saturating counters configure the PHT (Pattern History Table) which is indexed by the lower branch instruction address bits. Gshare predictor[3] is widely used in current processors. Gshare predictor uses the exclusive OR of GBH (global branch history) and branch address to make the PHT index.

One of the main reasons of miss-predictions is conflict aliasing, which is caused by the different branches accessing the same PHT entry. Major proposed predictors use global history to utilize the correlation among recently executed branches. Despite the steady improvements that have been made, it is difficult to completely avoid conflict aliasing, hence many branches are still miss-predicted [10].

This paper analyzes the behavior of branch predictor, and focuses on the miss-predictions bias. Miss-prediction bias means that miss-predictions of a few branches occupy a large rate of all miss-predictions. We propose a novel branch prediction mechanism by utilizing miss-prediction bias. Proposed mechanism is attached to a conventional branch predictor, and solves the conflict aliasing by preparing different PHT entry for each miss-prediction biased branch and by utilizing local history of miss-prediction biased branches.

The rest of this paper is organized as follows. Section 2 reviews the current branch predictors. Section 3 shows the characteristics of miss-prediction bias by using conventional predictors. Section 4 explains our proposal which targets for miss-prediction biased branches. Section 5 describes the evaluation of our proposal by simulation on SimpleScalar Tool Set[15]. In this simulation, our mechanism is attached to Combining, Bimode, Bimode-Plus, Agree, Hybrid and TAGE predictors. Section 6 and 7 discuss conclusion and future research.

2. Related Work

Many current branch predictors can be explained by extending the base predictors to reduce miss-predictions. The widely used base predictors are Bimodal and Gshare predictors. Several predictors using this approach are shown as follows.

Combining[3]: Combining predictor consists of a Bimodal predictor which works well for local history and a Gshare predictor which works well for global history. A selector is constructed by 2 bits saturating counter to select the result from either Bimodal predictor or Gshare predictor.

Bimode[4]: Bimode predictor prepares two Gshare predictors, one for the branches biased toward Taken (Taken Gshare predictor), and the other for the branches biased toward NotTaken (NotTaken Gsahre predictor). Then, Bimode predictor uses ChoicePHT to choose the result from two Gshare predictors. Bimode predictor can reduce conflict aliasing by dividing biased branches into different PHTs.

Bimode-Plus[7]: Some branches are strongly biased toward one direction (Taken or NotTaken) until the program finishes. Bimode-Plus predictor provides a Bias Table which keeps Taken bit and NotTaken bit to detect the branches which are strongly biased toward Taken or NotTaken. When the strongly biased branches are detected, the predictor uses

Table 1: Processor configuration

Pipeline	5 stages: 1 Fetch, 1 Decode, 1 Execute, 1 Memory Access, 1 Commit
Fetch,Decode	4 instructions
Issue	Int: 4, fp: 2, mem: 2
Window	Dispatch queue: 256, Issue queue: 256
BTB	2K-entry 4-way associative BTB, 32-entry RAS
Memory	64KB, 4-way associative, 1-cycle instruction and date caches 2MB, 8-way associative, 10-cycle L2

Bias Table without using the result of Bimode predictor nor updating into Bimode Predictor. By this way, Bimode-Plus predictor reduces the conflict aliasing between the strongly biased branches and normal branches.

Agree[6]: Agree predictor keeps the Taken biased bit and NotTaken biased bit in BTB. PHT keeps the result whether the prediction is same to the biased bit. When the branch result is same to the biased bit, the entry of PHT is incremented, otherwise it is decremented. Exclusive OR of the biased bit and the PHT result is used for prediction.

Hybrid[5]: Hybrid predictor keeps several predictors (2bc, Gshare, GAS, AVG) to predict the branch. BTB keeps the result of each predictor, and is used at prediction to select the most accurate one among the several predictors.

TAGE,L-TAGE[8], [9]: These predictors use PPM (prediction by partial matching) to search the patterns in the branch direction history[11]. This method has 4 PHTs which are accessed by the exclusive OR of different parts of GBHR and branch address. Every PHT entry has a tag (a part of branch address) to protect the conflict. Hence, this method uses the pattern of GBHR to improve the prediction accuracy.

3. Miss-Prediction Bias

We observe that miss-predictions of a few branches occupy a large rate of total miss-predictions on conventional predictors. This section shows the behavior of miss-prediction bias. Simulation is performed by using Bimode predictor on SimpleScalar Tool Set. Table 1 shows the processor configuration. Instruction set is PISA and benchmarks are bzip, gcc, gzip, mcf, parser, twolf, vpr and vortex from SPECint2000.

In experiments, top 8 and 16 miss-prediction branches are observed. We call these branches miss-prediction biased branches. Figure 1 shows the miss-prediction rate of top 8 and 16 miss-prediction biased branches to total miss-predictions. The experiments run 100M instructions ranged by 20M. The predictors size is set to 8KB, 16KB and 32KB.

In Figure 1, the horizontal axis means the executed instructions, and the vertical axis represents the rate of miss-predictions at the miss-prediction biased branches to total miss-predictions.

Clearly, Figure1 shows that miss-predictions at top 8 miss-prediction biased branches occupy more than 70% of total miss-predictions, and those of top 16 miss-prediction biased branches occupy more than 80% of total miss-predictions.

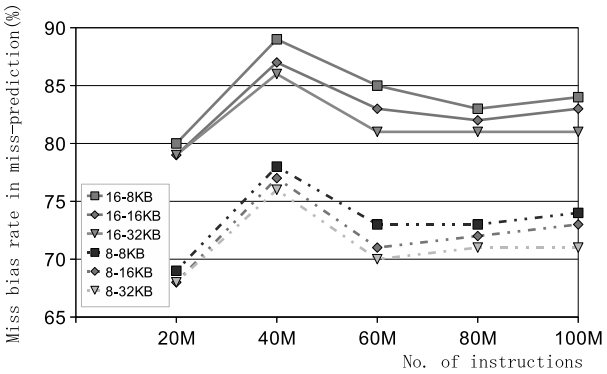


Fig. 1: Miss-prediction bias rate in Bimode predictor.

4. Proposed Predictor Attached on a Conventional Branch Predictor

The behavior of miss-prediction biased branches has been demonstrated in the last section. It means that a small number of branches cause most of miss-predictions. This section proposes a novel predictor to utilize this behavior. One of the major reasons for miss-predictions of many predictors is the conflict aliasing that the different branches accessing the same PHT entry. Our approach to solve this problem is to allocate different PHT entry for each miss-prediction biased branch, and to utilize local history for each miss-prediction biased branch without conflict aliasing. The proposed mechanism can be attached to any kind of conventional branch predictors.

Figure 2 shows a block diagram of our proposal. In this proposal, MBD (Miss Bias Detector) and LHBP (Local History Branch Predictor) are attached on a base predictor. MBD is used to detect miss-prediction biased branches. LHBP is used to predict the miss-prediction biased branches by using local history.

The prediction flows as follows. At first, miss-predictions of the base predictor are counted at the extended BTB (EBTB). When the count of miss-predictions exceeds the threshold, the branch is recognized as a miss-prediction biased branch, and its local history is stored into MBB (Miss Bias Buffer). Distinct LPHT entry is assigned for each miss-prediction biased branch to predict the branch direction based on the local history. At last, the Selector selects the result from either LHBP predictor or the base predictor as the final result.

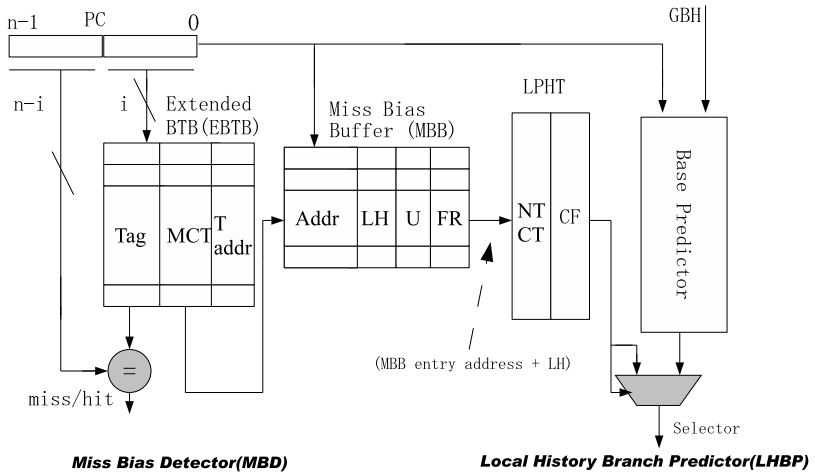


Fig. 2: Block diagram of proposed branch predictor

4.1 Detection of Miss-Prediction Biased Branches by MBD

MBD is composed of EBTB and MBB. EBTB is used to detect miss-prediction biased branches. EBTB is an extended version of BTB by keeping a MCT(Miss Counter) on every entry. MCT is a saturating counter for keeping the miss-prediction count of the base predictor. EBTB also keeps a Tag and branch target address (T addr) as conventional BTB. MBB is composed of Addr, LH, U and FR for keeping the information of miss-prediction biased branches. Addr keeps the address of miss-prediction biased branch, LH is a shift register for keeping the local history, U is a use bit for marking whether the MBB entry is used, and FR (Failure Rate) keeps the difference of miss-prediction count between LHBP and the base predictor.

Since the top 8 or 16 miss-prediction biased branches occupy most of total miss-predictions, MBB size is set to 8 or 16, and proposed predictor targets for top 8 or 16 miss-prediction biased branches.

The action of MBD includes registering the miss-prediction biased branches and updating the information of the miss-prediction biased branches. The action of MBD is explained as follows.

Registration of MBB : When the branch is committed, the branch address is associated in EBTB. If EBTB hits and miss-prediction occurs, EBTB's MCT is incremented, else if EBTB misses, Tag is changed as conventional BTB and the MCT is set to 1. If MCT arrives at the threshold, the branch address is registered into MBB, and corresponding EBTB entry is reset.

At registering to MBB, the branch address is checked whether it is already existed in MBB. If the address is not existed in MBB, it is registered into the entry whose U bit is 0. Then, the corresponding U bit is set to 1. If all of MBB's U bits are 1, the predictor uses LRU logic to search for the Least Recently Used entry, and registers the new branch address into the entry. The U bit is set to 1.

Update of MBB : When a branch is committed, the branch address is associated to MBB. If the branch address is found in MBB, the direction of the branch is shifted into LH. FR is used to denote whether the entry of LHBP is effective or not. FR is incremented when the predicted result of LHBP is incorrect and the predicted result of base predictor is correct. It is decremented when the predicted result of LHBP is correct and the predicted result of base predictor is incorrect. By this way, FR keeps the difference of miss-prediction count between LHBP and base predictor. When FR arrives at the threshold, it is recognized that the entry of LHBP is worse than base predictor. Then the LHBP entry is reset.

4.2 Prediction by LHBP

LHBP is a predictor targeted for the miss-prediction biased branches registered in MBB. Because the number of miss-prediction biased branches is small, LHBP uses local history of the miss-prediction biased branches for making PHT index. For avoiding the conflict among miss-prediction biased branches, distinct LPHT entry is prepared for each miss-prediction biased branch. Namely, LPHT entry is indexed by combining MBB entry address and its Local History. Hence, there is no conflict on the LPHT.

LPHT entry size is decided by the product of MBB entry size and local history length. Thus, when the local history length is n and the number of MBB entry is m, LPHT has $m * 2^n$ entries. Each entry of LPHT consists of NTCT and CF. NTCT is a 2 bits saturating counter for keeping the behavior of the branch. NTCT is incremented when the branch is Taken, and decremented when it is NotTaken. CF

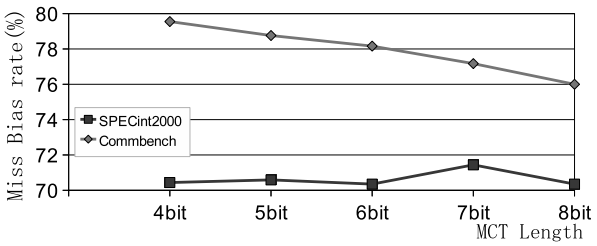


Fig. 3: Miss detection rate of biased branch to MCT length.

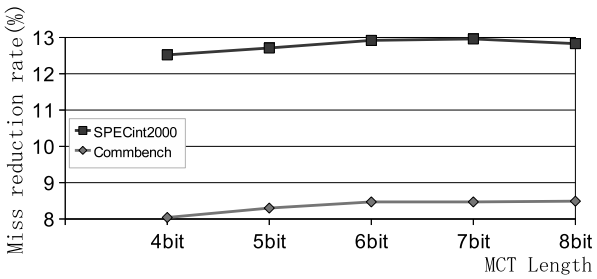


Fig. 4: Miss-prediction reduction rate to MCT length.

is the confidence of the LPHT entry, which uses 2 bit Miss Resetting Counter [12], [13], [14] of threshold at 3, because Miss Resetting Counter is known as the best method for increasing PVN(Predictive Value of a Negative Test)[12].

Update of LPHT: When the branch is committed, the branch address is associated to MBB. If the branch exists in MBB, the branch result (Taken/NotTaken) is used to update NTCT of the corresponding LPHT entry. When the LHBP prediction is correct, the CF is incremented, otherwise CF is reset to 0.

Branch Prediction: When a branch instruction is fetched, the branch address is associated to MBB. If the branch exists in MBB, the corresponding NTCT and CF values in the LPHT entry are used for branch prediction. At the same time, the base predictor predicts too. Then, Selector selects the result from either the base predictor or LHBP. If the CF arrives at the threshold, the predicted result of LHBP is selected, otherwise, the predicted result of the base predictor is selected.

5. Evaluation

5.1 Analysis of the Component Size

To evaluate our proposal, the optimum component size must be discussed.

Here we discuss MCT Length, MBB and LPHT entry size. MCT length is important for deciding the miss-prediction biased instruction. MBB entry size and LPHT entry size are important for increasing prediction accuracy. This section discusses the optimum component size by

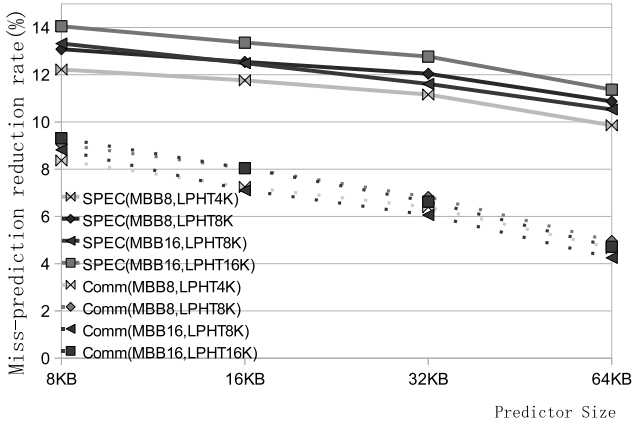


Fig. 5: Miss-prediction reduction rate to MBB and LPHT size (Bimode).

experiment. The experiment is performed on SimpleScalar Tool Set under the processor configuration given in Table 1. Benchmarks are bzip, gcc, gzip, mcf, parser, twolf, vpr, vortex of SPECint2000 and drr,reed_dec,reed_enc,rtr,zip_enc of CommBench [16]. Instruction set is PISA.

MCT Length: MCT is a saturating counter for counting the miss of BTB entries to detect the miss-prediction biased branches. For determining the MCT size, experiment is performed on the organization of Bimode Predictor as a base predictor with 8 entries of MBB and 8K entries of LPHT by ranging MCT length from 4 bits to 8 bits.

Figure 3 shows the rate of detected miss-prediction biased branches to total miss-predictions by varying the MCT length. Vertical axis is the average rate of miss detections. Figure 4 shows the reduction rate of miss-predictions. Vertical axis is the average reduction rate of miss-predictions.

By analyzing Figures 3 and 4, about 70% of miss-predictions can be detected and about 10% of miss-predictions can be reduced by our proposal. Besides, the miss-prediction reduction rate does not change so much by varying the MCT length. Hence, 4 bits MCT length is used in the following experiments in order to minimize hardware costs.

MBB and LPHT Entry Size: LPHT entry size is defined by the product of number of MBB entry and local history length. Generally, larger MBB size can store more miss-prediction biased branches, and bring better performance. Larger LPHT entry size can store longer local history, and bring more accurate prediction. In this experiment, MBB and LPHT entry size is set to (8,4K) with 9 bits LH, (8,8K) with 10 bits LH, (16,8K) with 9 bits LH, and (16,16K) with 10 bits LH. Bimode Predictor are used as base predictors. Figure 5 shows miss-prediction reduction rate by attaching our proposal to the base predictors. Vertical axis shows the average reduction rate.

By analyzing Figure 5, the difference of miss-

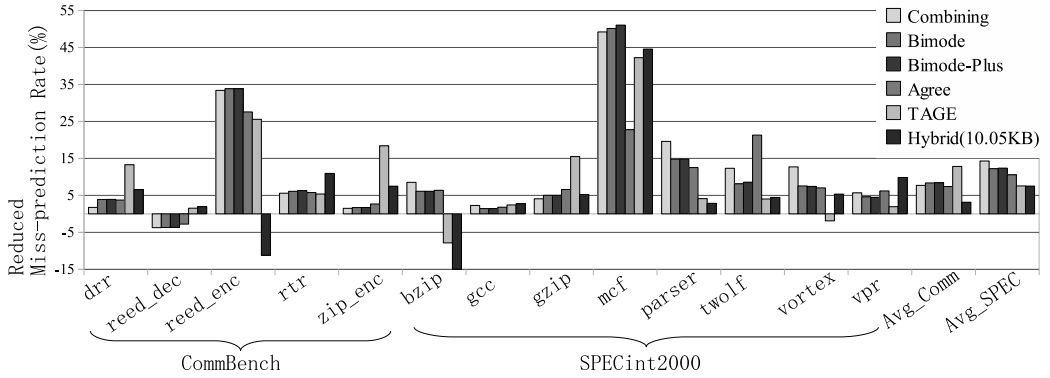


Fig. 6: Miss-prediction reduction rate using 8KB predictor.

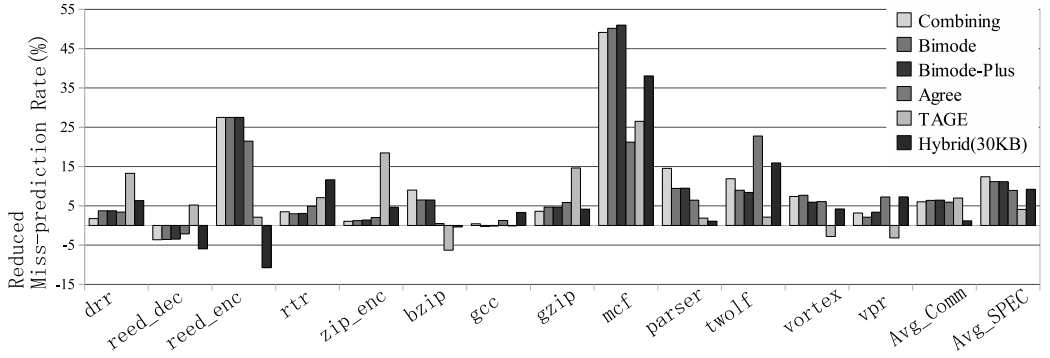


Fig. 7: Miss-prediction reduction rate using 32KB predictor.

prediction reduction rate between the largest setting (MBB=16,LPHT=16K) and the smallest setting (MBB=8, LPHT=4K) is only 2% in SPECint2000 and 1% in Comm-Bench with Bimode predictor. Hence, considering the hardware cost, the smallest setting (MBB=8, LPHT=4K) is used in the following experiments.

5.2 Miss-Prediction Reduction Rate and IPC Performance

We measure the performance of our proposal on several base predictors (Combining, Bimode, Bimode-Plus, Agree, Hybrid and TAGE predictor).

Figures 6 and 7 show the miss-prediction reduction rate of our proposal to the 6 kinds of base predictors sized of 8KB and 32KB. Horizontal axis shows the benchmark and vertical axis shows the miss-prediction reduction rate.

The results show that our proposal can reduce miss-predictions in 6 kinds predictors on average. When the base predictor size is 8KB, our proposal can reduce more than 7% of miss-predictions on average to the base predictors. When the base predictor size is 32KB, our proposal can reduce more than 5% of miss-predictions on average.

Figures 8 and 9 show the IPC performance enhancement of our proposal to the 6 kinds of base predictors sized of 8KB and 32KB. Horizontal axis shows the benchmark and vertical axis shows the IPC performance enhancement. The results show that our proposal can improve IPC about 2% on average to the base predictors.

In these experiments, we find Combining, Bimode, and Bimode-Plus predictors work similarly, because they use gshare and bimodal predictor. We find that there are several cases to prevent reducing the miss-predictions on our proposal. One is that some benchmarks have already good prediction on base predictor like reed_dec. In this case, our proposal can not further improve the prediction. Another is that the benchmark has a lot of miss-prediction branches like gcc. Our proposal does not improve because MBB size is not big enough to catch the miss-prediction bias branches correctly. When the branch predictor has several kinds of branch predictors like Hybrid Predictor, our proposal can not bring a large reduction of the miss-prediction. This is because Hybrid predictor can adopt to the branch behavior by changing the branch predictor. Our proposal can not catch the branch behavior so early.

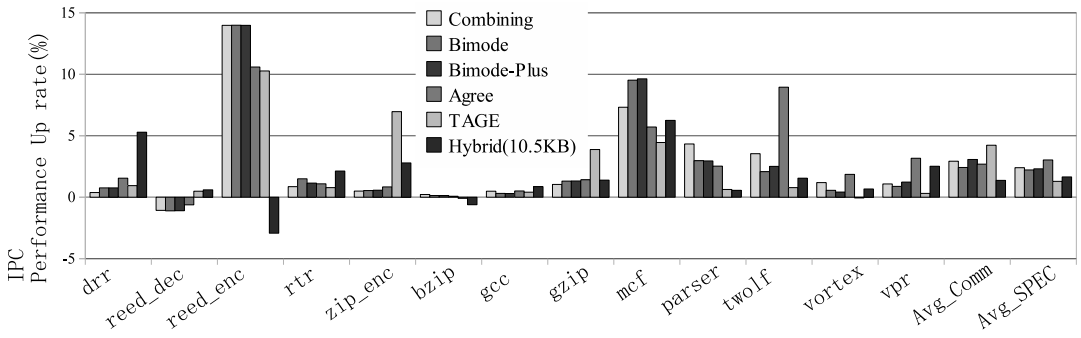


Fig. 8: IPC performance enhancement to 8KB predictor.

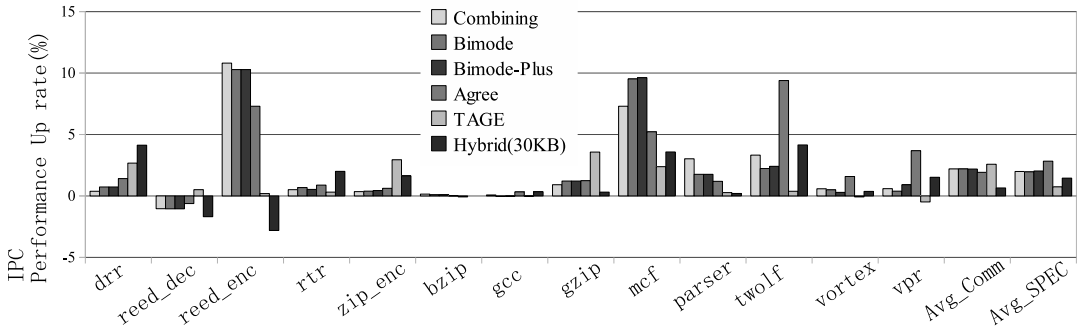


Fig. 9: IPC performance enhancement to 32KB predictor.

6. Discussion

6.1 Factors for Miss-Prediction

We think that major factors for miss-predictions are random behavior, conflict aliasing and undetectable branch patterns.

In order to improve the prediction accuracy, improvement of the latter two factors is important. Our proposed predictor aims to reduce conflict aliasing by detecting miss-prediction biased branches and preparing dedicated predictor for them by using local history. A number of predictors have been proposed to reduce conflict aliasing as explained in chapter 2. But, these predictors do not utilize local histories of miss-predicted branches. The key issue of our proposal is to utilize local history of miss-prediction biased branches, which makes it possible to detect patterns in the local history. The effectiveness of our proposal is proved by the experiment that the longer local history brings better performance, and prediction accuracy is improved by attaching proposed mechanism to conventional predictors.

6.2 Discussion on Hardware Costs

This subsection discusses hardware cost of our proposal by measuring the required memory size. EBTB uses the current BTB's association port. Hence it does not need to add any association port. Every entry in EBTB has additional 4 bits MCT and 2K entries, so additional hardware cost is only 1KB memory in BTB. MBB has 8 entries, every entry has 32 bits Addr, 9 bits LH, 1 bit U and 7 bits FR. So, MBB is a small CAM of 392 bits total. LPHT has 4K entries, every entry has 2 bits NTCT and 2 bits CF. So LPHT is a 2KB memory total. Hence, added hardware is small enough in our proposal.

Here we use MPKI (miss-predictions per kilo instructions) as a measure. Table 2 shows the MPKI of 6 kinds of base predictors (Combining, Bimode, Bimode-Plus, Agree, TAGE and Hybrid). The MPKI is the average of SPECint- 2000 and CommBench, where the base predictor size is 10.5KB, 17.75KB, 30KB, and 60.5KB in Hybrid Predictor. and the base predictor size is 8KB, 16KB, 32KB, and 64KB in other predictors,

Table 2 shows that our proposal can reduce MPKI than base predictors. In SPECint2000, our proposal attached on 8KB sized base predictor (Combining, Bimode, Bimode-Plus and Agree predictor) can achieve the same MPKI as 64KB sized base predictor. Experiment results show the same tendency in CommBench. And attaching proposed mechanism to smaller Hybrid and TAGE predictor can achieve the same MPKI as the larger predictor too.

Table 2: Result of miss-predictions per kilo instructions (Average)

Predictor	Specint2000				CommBench			
Size	8KB	16KB	32KB	64KB	8KB	16KB	32KB	64KB
Combining	5.53	5.30	5.06	4.84	7.34	7.20	6.96	6.67
Combining+proposal	4.76	4.60	4.43	4.30	6.48	6.43	6.35	6.18
Bimode	5.40	5.14	4.93	4.73	7.34	7.03	6.84	6.52
Bimode+proposal	4.76	4.55	4.38	4.28	6.50	6.37	6.27	6.15
Bimode-Plus	5.36	5.12	4.90	4.71	7.29	7.00	6.82	6.51
Bimode-Plus+proposal	4.71	4.50	4.35	4.24	6.45	6.34	6.23	6.13
Agree	6.71	6.49	6.35	6.21	7.51	7.44	7.35	7.20
Agree+proposal	5.94	5.79	5.69	5.61	6.78	6.88	6.82	6.75
TAGE	4.86	4.34	4.34	4.04	7.91	7.32	6.99	6.54
TAGE+proposal	4.41	4.15	4.08	3.90	6.81	6.55	6.37	6.09
Size	10.5KB	17.75KB	30KB	60.5KB	10.5KB	17.75KB	30KB	60.5KB
Hybrid	7.50	6.23	5.73	5.32	7.77	7.00	6.76	6.51
Hybrid+proposal	6.92	5.76	5.27	5.03	7.40	6.76	6.60	6.39

7. Conclusion and Future Work

Improving branch prediction accuracy is important for the modern processors which exploit instruction level parallelism by deeper pipeline and wider instruction issue width. This paper proposed a novel branch predictor for improving prediction accuracy by utilizing the behavior of miss-prediction biased branches. This predictor detects the miss-prediction biased branches and predicts the branch direction by using a local history based predictor attached on the base predictor.

This proposal is evaluated by experiment on the SimpleScalar Toolset. Our proposal is attached to Combining, Bimode, Bimode-Plus, Agree, Hybrid, and TAGE predictors. Miss-prediction reduction rate to the base predictors is evaluated. The results of experiment show that our proposal reduces the miss-predictions about 10%, and increases performance about 2% at SPECint2000. It also reduces the miss-predictions about 7%, and increases performance about 1% at Commbench.

Although our proposal reduces average miss-predictions to the conventional predictors, miss-predictions increases in some benchmarks. It must be detected and improved in the future work.

8. Acknowledgment

This work is supported in part by Scinics Co.,Ltd.

References

[1] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer, and J. Joyner. "POWER5 System Microarchitecture," IBM Journal of Research and Development, Vol 49, No 4/5, May 2005.

[2] J.E.Smith, “A Study of Branch Prediction Strategies,” Proc. of 8th ISCA, pp.135-148, 1981.

[3] S.McFarling, “Combining Branch Predictors,” Technical report TN-36, Digital Western Research Laboratory,1993.

[4] Chih-Chieh Lee, I-Cheng K. Chen and Trevor N. Mudge, “The bi-mode branch predictor,” MICRO97, pp.4-13, Dec.1997.

[5] M.Evers, P-Y.Chang and Y.N.Patt, “Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches,” ISCA 1996, pp.3-11, May 1996.

[6] E.Sprangle, Robert S. Chappell, Mitch Alsup and Yale N. Patt, “The Agree Predictor: A Mechanism for Reducing Negative Branch History,” ISCA 1997, pp.284-291, June 1997.

[7] K.Kise, T.Katagiri, H.Honda and T.Yuba, “The Bimode-Plus Branch Predictor,” IPSJ Trans.ACS-10, pp.85-102,2005.

[8] A.Seznec, “The L-TAGE branch predictor,” The 2nd JILP Championship Branch Prediction Competition (CBP-2), vol.9, May 2007.

[9] M.pierre, “A PPM-Like, tag-based predictor,” The 1st JILP Championship Branch Prediction Competition (CBP-1), vol.7, April 2005.

[10] L.Porter and D.M.Tullsen, “Creating Artificial Global History to Improve Branch Prediction Accuracy,” ICS’09, pp.266-275, 2009.

[11] I.-C.K.Chen, J.T.Coffey and T.N.Mudge, “Analysis of branch prediction via data compression,” ASPLOS-VII, pp.128-137, Oct.1996.

[12] H.Akkary, S.T.Srinivasan, R.Koltur, Y.Patil and W.Refaai, “Perceptron-Based Branch Confidence Estimation,” HPCA-10, pp.265, Feb.2004.

[13] E.Jacobson, E.Rotenberg and J.Smith, “Assigning Confidence to Conditional Branch Predictions,” MICRO96, pp.142-152, Dec.1996.

[14] Y.Ninomiya and K.Abe, “Power Reduction Based on Prediction Confidence Using a Perceptron Branch Predictor,” SACSIS2009, pp.327-334, May.2009.

[15] D.Burger and T.M.Austin, “The SimpleScalar Tool Set, Version2.0,” Technical Report, University of Wisconsin-Madison Computer Sciences Dept, July 1997.

[16] T.Wolf and M.A.Franklin, “CommBench - a Telecommunications Benchmark for Network Processors,” ISPASS-2000,Austin,TX, pp.154-162, April 2000.

A Modular Processor Architecture for High-Performance Computing Applications on FPGA

Fritz Mayer-Lindenberg
Institute of Computer Technology
Technical University of Hamburg-Harburg
mayer-lindenberg@tuhh.de

Abstract.

This paper presents the architecture of an embedded processor for numeric applications that is tailored to the resources of current FPGA chips but not limited to FPGA based applications. The architecture implements several non-standard and even novel features. The most basic feature is to modularize the processor into a standard control core and the ALU, allowing the same controller to be used for several ALU circuits and thus exploiting the capability of the FPGA to support all kinds of data types. The controller also provides memory control for a separate data memory attached to the ALU. Another basic feature related to the limited FPGA resources is to realize techniques such as multi-threaded control to fill the ALU pipeline otherwise found on high-performance processor chips only with a low circuit complexity. The controller is about half as complex as a simple floating-point ALU. Both together consume about 2500 LUT cells of an FPGA only. As an example for an ALU using more of the arithmetic resources of the FPGA we present a vector ALU operating on 144-bit data words encoding real or complex vectors and quaternions. For the processor based on this ALU the control overhead is below 10%. The processors are interfaced to private program and data memories on the FPGA, and to external memory via an extra peripheral device on the FPGA.

Keywords*:* Harvard architecture, fine-grained multi-threading, soft caching, block floating point vector ALU, network-on-FPGA

Introduction.

Modern FPGA chips contain tens of thousands of single output bit cells as well as hundreds of hardwired complex arithmetic building blocks such as parallel 18-bit multipliers, and embedded memory blocks. It is thus not astonishing that high-end FPGA chips have been proposed for supercomputing applications [1,8]. For a commercial FPGA system for HPC integrated with PC hardware see [11]. The use of the FPGA for processing numbers demands for computational circuits, including scalar and vector floating-point units, to be implemented. Following [2], we take the approach to implement individual sequential controllers for these, in other words networks of processors, in contrast to coprocessing schemes. To efficiently use the FPGA resources, the controllers must be simple, however. Conventional CPU designs like [4] supporting a large memory space with caching and memory management and using a scalar general-purpose ALU are quite complex and involve large control overheads. They communicate with their on- and off-chip memories and peripherals via parallel memory buses that don't support more than a few processors. We therefore set out for a processor design with lower control overheads, allowing more processors to be implemented on an FPGA chip to achieve a higher arithmetic performance, and also one that would facilitate the use of more powerful application-specific ALU circuits.

Actually, all of the current highest-speed processor architectures achieve their performance through the use of more complex ALU circuits using multiple pipelines and applying SIMD processing. The highest performance DSP family at the time being, the TMS320C667x family from Texas Instruments uses two arithmetic units each starting up to 4 single precision floating point multiply and 4 such add operations in a single cycle, using a VLIW instruction set architecture [12]. The highest performance chip of this family is the 8-core TMS320C6678 performing at a peak rate of 128GFLOPs in single precision at a clock rate of 1GHz. The shader circuits of the recent GPU chips perform similar sets of scalar operations in parallel at similar rates and use extensive SIMD on top [13].

In the present work, the 45nm Spartan-6 family of FPGA chips from Xilinx [3] has been used to implement the new processor architecture. Spartan-6 is a low cost family within the FPGA world. The most complex chips of this family is the XC6XLS150T which contains about 92k 1-bit cells, 180 multipliers and 258 separate RAM blocks of 2k bytes each, enough to implement about 24 scalar floating point processors or 4 vector processors. It costs about as much as the TMS320C6678 yet performs about 10 times slower if scalar single precision operations are considered. This reflects the configuration overheads inherent in the FPGA architecture. If the extended single precision implemented in the FPGA processor is needed, the FPGA becomes more competitive. High-end FPGA chips can deliver a much higher performance than the Spartan-6 FPGA, yet hardly a better price by performance ratio.

Processor designs on FPGA have been used for a long time, in spite of the involved configurations overheads, and are easy to achieve as far as the basic function of a programmable processor is concerned, and are used in books and courses on digital design [10]. Some processor designs addressing the FPGA and trying to exploit its flexibility use features such as mutli-threading and VLIW that are also important in the present design [14,15,16]. Its unique features are to closely adapt to the FPGA resources for maximum efficiency, including an ALU design adapted to the avaiilable lookup table to multiplier ratio.

Clock rates achieved for FPGA based processors are between 20 and 200MHz. [7] reports on a processor design running at more than 300MHz. The present architecture has several predecessors [9,10]. It was first implemented at a clock rate of 100MHz and is being updated to 200MHz by extending the controller and ALU pipelines. Before entering in the details of the new architecture, we make a few remarks on the resources for implementing on FPGA that have influenced its design.

1) The hardwired memory blocks are the natural choice to implement on-FPGA memories for the processors (e.g., cache memories). Their maximum access frequency then places an upper limit on the processor clock frequency at about 300MHz, assuming caches to be accessed at the CPU clock rate. If an FPGA is filled up with processors, just a few k words remain for each, however. This actually sets an extra premium on not implementing too many processors but rather a few only using complex ALU circuits instead.
2) The FPGA block RAM has some characteristics and benefits that can be exploited for the processor design. First, the word width of the memory block can be configured to 9, 18, or 36 bits and hence provides some extra bits in comparison to the common 16 and 32 bit instruction and data widths found in hardwired processor designs. Second, the memory blocks are dual ported and hence allow two independent accesses to be performed simultaneously.
3) The hardwired arithmetic circuit functions include an 18-bit signed parallel multiplier and a 48-bit adder-subtractor. These don't suffer from the FPGA configuration overheads and operate at rates of up to 250MHz using their built-in output registers. They are significantly faster than multipliers built from individual FPGA cells. In order to fully exploit the FPGA resources, integer data should be rather 18-bit than 16-bit, or 35-bit rather than 32-bit.
4) In order to achieve clock rates close to the maximum rates of the memory blocks and the multipliers pipelining must be applied to the processor design. Consequently, ALUs must be pipelined and need control facilities for filling their pipelines from several threads unless the algorithms to be executed happen to achieve this from a single thread [6].
5) The Spartan-6 family and some others offer a feature called distributed RAM which consists in configuring an individual FPGA cells that otherwise implements Boolean functions through a look up table (LUT) as a tiny RAM with up to 64 addressable single-bit locations. This feature permits to realize register files with about the same cell count as individual registers. Register files are most useful for implementing addressable multi-port data register banks and dedicated return stack memories. Multiple threads can be supported by performing zero overhead context switches, subdividing large register files into sub files associated to and selected for the threads.

The controller architecture

We discuss the processor architecture in terms of architectural features. The first two of them are concerned with the diversity of data types supported by the FPGA.

AF1. This first architectural feature is to subdivide the programmable processor reading from an instruction list in memory to control its ALU for the different program steps into a controller circuit and the ALU. The controller addresses the instruction memory (IRAM) and reads out instructions, implements the control flow, computes data addresses and commands IO and memory accesses.

AF2. The ALU and the controller will get a memories of their own (Harvard architecture). The controller controls the data memory (DRAM) of the ALU yet doesn't access the data. The data memory and the data registers provided for the ALU may have an arbitrary width that may be different from the width of the instruction memory.

AF3. The instruction memory is directly interfaced to the controller, as is the data memory to the ALU. Both are implemented with FPGA memory blocks. Memory addresses and write data are applied synchronously. No extensions of the memory interfaces are provided to decode IO circuits or external memory locations.

AF4. The address spaces for the instruction an data memories are 15-bit only. The program counter used by the controller is even reduced to 14-bit.

AF5. The controller provides an input port with DMA support that connects to peripheral devices including an external memory controller. DMA input into the DMEM is supported, too.

The latter features serve to simplify the interfacing of the processor and of the address generation. In particular, no automatic cache control and no memory management are used. Instead, software controlled caching is implemented by setting up a DMA from an external memory controller and commanding it to send an instruction or data packet.

AF6. The controller also performs data accesses to the IRAM. Controller registers and data are 18-bit. Data accesses performed by the controller occur on the second port of the dual ported IRAM. Thus a single memory is accessed both for instructions and controller data, yet using separate ports as in a Harvard computer.

The following is a performance feature permitting controller instructions like jump instructions of memory accesses to occur in parallel to ALU operations. It also allows to adapt to all kinds of ALU circuits.

AF7. The controller uses 18-bit instruction codes. These are extended by extra instruction bits passed to the ALU by widening the read port of the IRAM, thus using VLIW. For the ALU discussed below, the widening is to 36-bit, but larger instructions words could be used as well.

AF8. The controller executes up to 4 control threads, Zero overhead context switches are implemented through register bank switches in response to an instruction bit.

The controller provides 8 register named I0...I7 and a single control flag used as branch condition. Table 1 lists the controller instructions. They are encoded in the lower 17 bits of the instruction word. The remaining bit, the 'x' bit, is used to control the context switches.

Table 1: The controller ISA

jp addr	unconditional absolute jump
call addr	subroutine call
ret, it-=offs	return from subroutine
cjf addr	conditional absolute jump for F=1
ijnzt addr	increment and jump if negative
it=im(a)	absolute register load
it=im(is-offs)	indirect register load
im(a)=it	absolute register store
im(is-offs)=it	indirect register store
it = n	load short constant
it+=n	add short constant
it*=n	multiply short constant
ir=it+is	add modulo 2^18, set F=msb
ir=it-is	subtract modulo 2^18, F=msb
ir=it&is	bitwise 'and' operation, F=parity
ir=it*is	multiply modulo 2^18
ir=it	register move
ir=time etc.	internal IO instruction group
ir=alu	input from ALU port, F=af
alu=ir	output to ALU
F=af	flag input from ALU
F=iordy(a)	external flag input from address a
it=io(a)	data input to register and to F
io(a) = it	output
dmaa=it	DMA input control group
dt=dm(a)	absolute load from ALU memory
dt=dm(is-offs)	indirect load, is++ for offs=0
dm(a)=dt	absolute store to ALU memory
dm(is-offs)=dt	indirect strore, is++ for offs=0
dt=dm(is-offs),dt'=dm(s)	dual indirect load
spare codes	reserved for ALU instructions

One remarks the small number of arithmetic and logic instructions. Even an add with carry is missing. The reason is that the controller is mainly involved in address computations while all numeric processing is performed by the attached ALU. Bitwise 'or' and 'xor' operations can be derived from the '+', '-' and '&' operations, if needed. The F bit is set by a preceding arithmetic or input instruction.

The 'io' instructions don't handshake and behave like memory accesses. They save the decoding of memory mapped IO and can be performed in parallel to memory accesses. Handshaking becomes available through the external flag input instruction. IO is further supported through two DMA input channels. Some predefined interfaces can be accessed through internal IO instructions. The 'time' register provides a time reference for real time applications. Also, there are predefined interfaces to send to or receive from an external host computer.

The 'dm' instructions are provided to control the data memory attached to the ALU. One is a dual read instruction using the second port of the dual ported DMEM and an extra pointer register for each thread that can be loaded with another internal IO instruction. ALU data can only be transferred via the 18-bit 'alu' port accesses which also don't perform handshaking. Data transfers with a numeric ALU can be combined with ALU conversion operations to and from 18-bit integers.

AF9. 18-bit integer codes converted by the ALU or to be converted into ALU data are stored in the controller memory and registers and are input and output there. ALU memory is reserved for the ALU data type. The same principle applies to the results of comparisons of ALU data. These are passed to the F bit of the controller.

In the 100MHz implementation, the controller pipeline has 4 stages computing the next fetch address, performing the instruction fetch, decoding and selecting operands, and executing and writing back the result. During the decoding of a jump instruction, the instructions following the jump is still fetched such that all jumps are delayed.

Each of the 4 contexts comes with its own set of registers and an 8-level return stack. Thus no time is lost to save and restore registers when a context switch occurs. A context switch propagates through all stages of the pipeline, shown here for the 4-level 100MHz pipeline:

```
Address   - |---K1---|---K2---| - -
Fetch           - - |---K1---|---K2---| - -
Decode                  - - |---K1---|---K2---| - -
Execute                         - - |---K1---|---K2---| - -
```

During the decode cycle for context 2 (K2) operands are selected from the registers of this context while the write back at the end of the execute cycle in K1 still occurs to a register in context 1.

Context switches are controlled in a non-standard way to never lose cycles through waiting and making sure that a sequence of ALU instructions can proceed at its maximum rate even if some cycles are executed by other threads. Context switches occur in three cases, under program control in response to the 'x' bit of the controller instruction, by activating thread 3 or on request from the ALU.

A context switch in response to the 'x' bit (AF8) in one of the threads 0-2 passes execution to another one of these if there is one that is ready to run. Thus with regular patterns of 'x' instructions the execution time can be smoothly distributed between the threads. The 'x' combined with the an external flag input instruction is decoded to let the calling thread become inactive until the external input gets active ('1'). If this occurs to thread 3, it immediately takes over due to its higher priority. A handshaking input thus takes two instructions:

```
X  F=iordy(a)   .. release thread, select iordy(a) to resume
   it=io(b)     .. after resuming, input from IO address b.
```

More than one ready signal can be selected to resume to implement alternative input. The signals waited for by some thread are continuously scanned.

AF10. Finally, the ALU can force a context switch to the thread encoded by its ACTX output by activating its FRC signal. Execution can thus be passed for just a few cycles to another thread that might start more ALU operations during this time. The generation of FRC is decoded by the ALU from its instruction bits.

The thread executing an ALU instruction with the 'x' bit set and implicitly or explicitly encoding the use of FRC, e.g. an add operation accessing some data registers like

XF d4=d6+d8 ,

is released after the start of the operation but resumed in response to FRC, typically at the earliest time at which the result of the operation can be used. For an ALU with a 4 stage execute pipeline, the executing thread would e.g. pause for 3 cycles and then continue.

The interface signals between the controller and an attached ALU are listed in table 2. They don't include an ALU instruction code which might be decoded by the ALU from the controller instruction word or be an extra instruction word read in parallel to the controller instruction. There is an 18-bit port for transferring data between the ALU and the controller (with or without a conversion to 18-bit integer codes). There is no handshake signal, however, for such exchanges. The controller instructions accessing the ALU port are simply read and executed in parallel to the transfer operation of the ALU.

Table 2: Interface signals to an ALU

IOUT	18-bit output to ALU
ALUIN	18-bit input from ALU
AFI	flag input from ALU
CTX	actual CPU context (decode cycle)
ACTX	desired output context for ALU
ENA,FRC,INH	context control signals
ARWD,ARWE	ALU register write control
ARRA	optional 4-bit ALU register address
FETCH	signals a valid instruction fetch

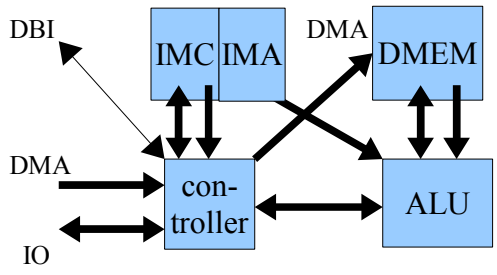


Fig.1: Modular processor

Fig. 1 shows a full processor based on the controller and an attached ALU. Finally, the controller is linked to a host controlled synchronous serial 5-bit bus via the DBI port. All processors implemented on the same FPGA are linked to this bus. The DBI interface implements the host interface of the controller, supports the download of programs and some simple debug facilities.

The controller is complemented by an external memory controller and interfacing components used to construct a data network within the FPGA, and a controller of the host interface bus already mentioned. External memory data from the memory controller enter into the DMA port. The controller connects via its IO port to up to 16 interfaces. The controller manages IO for the ALU data by first transferring them from the ALU to the controller or vice verse. Direct wide data IO from the ALU can be supported through corresponding ALU instructions.

AF11. The memory controller providing access to an external memory to the controller is an interface device that receives or sends blocks of data to the processor responding to command words from the processor. The command words identify the blocks within the external memory and typically contain the upper bits of its starting address. The lower address bits are generated by the memory controller. Only the memory controller is concerned with forming wide addresses for the physical memory; the controller only addresses the on-FPGA memory. The memory controller also converts between blocks of 18-bit data words from the controller and blocks of 16-bit words suitable for the external memory device.

Programs of unlimited sizes can be executed on the controller once software caching is used. It involves extra instructions commanding to the memory controller to send new blocks of instructions and for setting up and synchronizing DMA transfers. Even so this technique is considered superior to spending FPGA resources for automatic cache control. Software caching can be fully pipelined with program execution and keeps a deterministic timing for the latter. The extra instructions can be synthesized at compile time and in principle be generated automatically by a compiler.

AF12. Processors only communicate with each other through dedicated networking interfaces. For small numbers of processors, directional point-to-point interfaces attached to their IO ports suffice. For larger numbers, a bus interface component is being used, even allowing a processor to connect to several communication buses.

Processors on the same FPGA can't directly communicate by simply connecting their 18-bit IO input and output ports due to the uncorrelated execution of their instructions. At least a buffer register with handshaking signals is required. Moreover, the communicating processors can be far from each other on the FPGA such that large routing delays occur. The timing of the signals must then be reestablished by introducing registers on the way, and a multi-word receive buffer large enough to support the delay of the backward handshake. The resulting structure is implemented as a directional link component, using distributed RAM for the buffering.

AF13. Processors can also use their DMA port to receive data from a link or bus interface. Based on this, a processor can perform remote writes into the local memory of another processor to make some of its state visible to it.

The bus interface component is not part of the CPU design but essential for the on-FPGA networking of several processors. Besides the port by which it attaches to a processor, it provides input and output ports to the bus. The bus is formed by connecting a number of bus interface components (up to 16) by their bus input and output ports into a directed ring. The bus input ports are registered, and the ring actually is a ring register in which the data words stored in the registers advance by one step in every clock cycle. In contrast to a conventional bus, the ring hence transports several data words in parallel. It is a synchronous circuit using directional signals only. The data words on the bus are 24-bit words containing 18 bits of payload data, a 4-bit destination address and two bits required for managing the transfers. The destination address of a message is defined through a control output from the sending processor.

The V144 ALU

The first ALU combined with the controller as described above has been a floating point ALU operating on 45-bit data words. The 35-bit mantissa requires 4 18-bit multipliers only. It uses 4-level pipelines in the 100MHz implementation. Although the control overheads in a processor based on this ALU are quite acceptable 30% of the total of about 2500 LUT cells, additional overheads arise for the on-FPGA networking and the sharing of memory controllers. The control and communications overheads can be reduced by using a smaller number of processors equipped with complex (non scalar) ALUs. This approach is also capable of using more of the large numbers of multipliers offered on the most recent FPGA families.

There are several choices to control an ALU with multiple multipliers and adders all of which are supported by the controller. The simplest and most common one is to use SIMD operations. The most flexible one is to use VLIW control and a bank of registers with multiple write ports. The wide instructions and the complex register structure raise the control overhead, however. We opted for an ALU providing SIMD instructions and some special non-SIMD operations yet requiring a single extra instruction word only. While the scalar floating point ALU consumes 4 FPGA multipliers only, the vector ALU consumes 16 of them. The number of scalar processors that fit into a Spartan-6 FPGA is bound by the number of available LUT cells and can't exploit all multipliers, the number of processors based on the vector ALU is bound by the number of available multipliers. Only the vector ALU can fully exploit the combined performance of nearly all of the available multipliers. The smaller number of vector processors also reduces the communication overheads and somewhat simplifies the programming. The most recent FPGA chips pack still more multipliers in relation to the LUT count. Then a port of the present design becomes LUT bounded again. Then an extension to still wider data words can be considered, using SIMD on top of it similarly to what is done in recent GPU chips.

The V144 ALU uses a 144-bit data word holding a vector of 4 35-bit fixed point components that are encoded by an unsigned 34-bit number and a sign bit. There is a spare bit for each component that allows for temporary add overflows. The word (r,x,y,z) is also written as r+ix+jy+kz. They can be interpreted as real 4-vectors, as complex 2-vectors or as quaternions [5]. Sub types are the real and complex numbers r+ix, and real 3-vectors (0,x,y,z). The quaternion product is the bilinear mapping defined on pairs of the base vectors 1,i,j,k by

1*1=1, 1*i=i*1=i, 1*j=j*1=j, 1*k=k*1=k, i*i=j*j=k*k=-1,

and i*j=k, j*k=i, k*i=j, j*i=-k, k*j=-i, i*k=-j .

It extends the multiplications on the sub types of real and complex numbers and the product of a complex number and a complex 2-vector. The conjugate of the quaternion q=r+ix+jy+kz is defined by conj(q)=r-ix-jy-kz, extending the conjugation of complex numbers. Then

$$q*conj(q) = r^2+x^2+y^2+z^2 .$$

If $r^2+x^2+y^2+z^2=1$ then the mapping $v \rightarrow q*v*conj(q)$ is a rotation of the space of real 3-vectors. Hence unit quaternions can be used to parametrize rotations. The quaternion type thus summarizes several useful real and complex data types.

The V144 ALU implements the quaternion product through 4 instructions performing the multiplication of the different combinations of half words (one being the complex multiply operation), and each performing 4 real multiplies in parallel, and some add operations. It also provides more conventional SIMD type add operations and multiply and dot product operations both with a double precision result, and a radix-2 butterfly operation on complex 2-vectors. For the latter, a constant table is included hold-ing the twiddle factors, using a read-only memory block.

The ALU uses separate 18-bit instructions that are fetched along with the controller instructions. Separate sub fields of the ALU instruction are used to independently select ALU operations and transfer operations to the controller. Operands are selected from 16 144-bit data registers. Each context has its own bank of registers. The banks are subdivided to permit two parallel register writes. To transfer data between a data register and the controller, 8 transfer instructions are needed each moving an 18-bit sub field. The transfer operation of the ALU is used in parallel with an ALU port access by the controller. Data passed between the ALU and the controller are converted on-the fly between the sign+magnitude and the twos-complement formats. The shifting hardware transferring an ALU word in multiple 18-bit packets to the ALU port of the controller is also used to access and inspect the ALU registers

during single-stepping through a program. Table 3a lists the ALU operations and table 3b the transfer operations.

Table 3a: Arithmetic instructions of the V144 ALU

dr,dr'=ds*dt	SIMD product, double result, frc
dr=ds+dt	SIMD sum, first shift ds
dr=dt–ds	SIMD difference, first shift ds
drl=dtl*dsl	complex product L, single res., frc
drh=dtl*dsh	complex product H, single res.,frc
drh+=dth*dsl	quaternion product L, single
drl+=dth*dsh	quaternion product H, single
dr=(dt,ds)	dot product, double result, frc
dr+=(dt,ds)	accumulate dot product, frc
butterfly	uses twiddle factor from ROM
ds+dt',ds'*dt,dr=res	SIMD dot product, double, frc

Table 3b: Transfer instructions of the V144 ALU

shift dr	use shift count in control register
dshift dr	shift double to single
dr=dt	register move
dr = conj(dt)	complex/quaternion conjugation
dr=cpu	shift from CPU
cpu=dr	rotate into CPU

Some instructions include a multi-bit shift operation. The shift count is provided in a control register and is typically used for an entire 4n-vector composed of n 4-component data words. The shift operations combine with the add/subtract operations into floating point operations yet using the same exponent for all components of a vector. The 35-bit fixed point codes for the vector components can be considered as mantissas. In general, they can't be all normalized. During a multi-cycle vector operation, the shift count for a subsequent normalization is established automatically. Exponents for vectors are stored in the 18-bit controller memory. They are not part of the vector codes in the data memory. The computation of shift counts is left to the controller. This simplifies the ALU but also causes some overhead for scalar floating point operations. The use of exponents and shift counts is optional. Fixed point processing can be used instead. The SIMD product and dot product operations accumulate the products with the full (double) precision. No further support is provided to increase the precision.

The two complex vector components used as inputs to the butterfly operation are loaded as a single data word but have a large indexing distance in the input vector to the current FFT pass. In the next pass, the butterfly input needs to combine the outputs of two different butterfly computations in the current pass. Therefore the results of both butterflies are packed accordingly before restoring them to the memory locations previously holding their vectors. The bit-reversed address sequences and the twiddle factors for the different passes are selected by means of the control register. During a pass, an input vector is fetched and an output vector stored in every cycle, using the dual memory access operation of the controller in parallel to the butterfly instruction. During each pass, the occurrence of an overflow is tracked, and the load operations in the next pass are combined with a shift operation to take care of it.

The current V144 ALU uses 16 FPGA multipliers which are employed to close to 100% during an FFT or dot product computation. A stream of 3D vectors can be transformed by a fixed rotation continuously using 3 of the 4 multipliers. 10 V144 based processors running at 200MHz fit into an XC6SLX150T FPGA and yield a peak performance of 16 GFLOP/s of extended single precision block floating point operations. A variant of the V144 ALU is being developed using only 12 DSP modules (3 for a 34-bit multiply). Then up to 13 processors are expected to fit into the FPGA with a total peak performance of about 20GFLOP/s.

System applications and tools

The described processor architecture ir ready to be used on the Spartan-6 family of FPGA. The minimum configuration consists of a single controller with 1-2 memory blocks and some application specific peripherals attached to its IO port, and the host bus controller attached to get external control, program download and debug support via a UART interface or the JTAG interface of the FPGA. Larger systems are easily constructed from the processors and the interfacing building blocks. The tiny XC6SLX9 FPGA with about 5700 LUT cells can already hold a system of two scalar floating point processors linked to each other and to a memory controller. A similar system based on a Spartan-3 FPGA has been used to provide a low-cost micro processor lab. The lab uses assembler programming; a C compiler is available, too.

The current architecture implemented on Spartan-6 is supported by the π-Nets compiler environment described in [10]. π-Nets can support targets that are networks of processors. It supports multi-threaded applications and real-time control and can even be used to derive a suitable processor network from a multi-threaded program [2]. More programming tools are planned.

The main application of the new processor design is the design of an experimental parallel computer based on a network of XC6SLX150T chips. This system is intended to study and to develop tools and methods for FPGA based high-performance computing (HPC) applications. The individual FPGA nodes hold small networks of V144 based processors. External memory chips attached to the FPGA nodes are accessible by all processors via memory controller interfaces as described above. The FPGA chips are linked by high-speed serial interfaces which are transparently accessed from the on-chip networks using an additional networking interface module. The current system contains 20 FPGA nodes and hence up to 200 processors (depending on the configurations being used) with a peak performance of 320GFLOP/s. The peak performance of an individual processor can be achieved in some common applications.

The performance features such as single cycle execution of all CPU instructions including memory accesses and the execution of ALU instructions in parallel to them, the low complexity of the CPU, the inclusion of DMA and real time control, the capabilities of the two ALUs, and the ready-to-use interfaces and networking functions are attractive for other applications than HPC as well, in particular for embedded control and DSP applications.

Discussion

The proposed processor architecture takes several non-standard choices which derive from the particular requirements but which may raise doubts on its usability. We discuss some possible objections to our design and justify the design decisions taken.

The first objection may regard the very small address space of the processors which is below the 65k range of very early processors. The general trend has been to extend the logical address spaces to 32-bit addresses and beyond. A complex operating system or a GUI can hardly be realized this way, but also the intended HPC applications require the handling of large vectors and matrices.

The actual amount of addressable memory is not that small, however. The 15-bit controller addresses are word addresses and allow for up to about 65kBytes for the controller. The data memory is an extra address space on top of this which could be as large as 512kB for the V144 data type. These memory sizes are far from what would ever be realized within the FPGA for an individual processor. Large HPC data would have to be stored in external memory and only be cached in the DMEM.

Modern processors come with many registers, but the controller provides 8 registers only. Also, the return stacks are very small. There seems to be no room for stack frames.

For this objection one has to remark, that most of the registers are located in the ALU, e.g. 16 in the V144 ALU. The controller registers mostly serve as address pointers and loop counters. The controller has 8 registers for every context. The stack used by the 'call' and 'return' instructions is dedicated to holding return addresses and can't be used to pass parameters. A stack for passing parameters and holding local variables can be realized in IMEM or DMEM using any of the 8 pointer registers. The 'return' instruction optionally deallocates stack locations.

Why do the jump instructions use absolute addresses? The 'x' in every instruction appears to be a waste of instruction memory. Why is the DMA limited to inputs?

Absolute addresses save a bit of hardware. The input only DMA needs a single address register only; there is no counter as the last word of a block transfer is marked by a control signal. 'x' is not related to particular instructions and hence needs an extra field. Most choices in the instruction set and in the hardware design have alternatives and are to some degree arbitrary. Important is the decision to keep the controller hardware as simple as possible while providing maximum flexibiliy and efficiency of control for the attached ALU.

HPC applications usually operate on IEEE standard single and double precision floating point numbers. Wouldn't it be better to implement these in the ALU circuits? And why should one consider at all an exotic new processor architecture once there are ready-to-use high-performance chips?

The essence of the new architecture is the functionality of the controller and its interface to an ALU to form a full processor. The controller can be combined with an ALU for one of the standard types. The chosen types really exploit the FPGA resources while for the standard types the FPGA is much less attractive than some recent hardwired DSP chip. It depends on an analysis of the precision required for an HPC application whether the types optimized for the FPGA can be applied. Whether a processor is adopted by many users only partly depends on its technical benefits. Anyhow, the ideas in a published architecture can be taken up by other developments.

References:

[1] R. Baxter et al, Maxwell – a 64 FPGA supercomputer www.fhpca.org

[2] F.Mayer-Lindenberg, High-level FPGA Programming through Mapping Process Networks to FPGA Resources, ReConfig09, Cancun, 2009

[3] www.xilinx.com, Spartan-6 series user manuals

[4] www.xilinx.com, EDK with µBlaze processor

[5] A.Watt, 3D Computer Graphics, Pearson 1999

[6] Hennessy, Patterson, Computer Architecture, Morgan Kaufmann Pub. Comp.

[7]Andreas Ehliar, Performance driven FPGA design with an ASIC perspective, PhD thesis, Linköping 2009

[8] S.Craven, P.Athanas, Examining the viability of FPGA supercomputing, EURASIP J.Emb.Systems 2007

[9] C.Bassoy et al.,SHARF:An FPGA based Customisable Processor Architecture, FPL'09

[10] F. Mayer-Lindenberg, Dedicated Digital Processors, Wiley Interscience 2004

[11] www.sciengines.com, RIVYERA S6-LX150 system

[12] www.ti.com, TMS320C667x family documents

[13] www.amd.com, HD7000 GPU architecture

[14] R.Diamond, O.Mencer, W.Luk, CUSTARD -customisable threaded FPGA soft processor. FPL'05, 2006

[15] W.Chu et al, Customisable EPIC processor, design automation and test in Europe conference, 2004

[16] S.Wong, T.v.As, G.Brown, r-VEX: reconfigurable and extensible softcore VLIW processor, ICFPT'08

Field Programmable Gate Arrays for Computational Acceleration of Lattice-Oriented Simulation Models

A. Gilman and K.A. Hawick
Computer Science, Institute for Information and Mathematical Sciences,
Massey University, North Shore 102-904, Auckland, New Zealand
email: { a.gilman, k.a.hawick }@massey.ac.nz
Tel: +64 9 414 0800 Fax: +64 9 441 8181

April 2012

ABSTRACT
Field Programmable Gate Arrays (FPGAs) have attracted recent attention as accelerators for a range of scientific applications which had previously been only practicable on conventional general purpose programming platforms. We report on the performance scalability and software engineering considerations when FPGAs are used to accelerate performance of lattice-oriented simulation models such as complex systems models. We report on the specifics of an FPGA implementation of cellular automaton models like the Game-of-Life. Typical FPGA toolkits and approaches are well suited to the localised updating of the regular data structures of models like these. We review ideas for more widespread uptake of FPGA technology in complex systems simulation applications.

KEY WORDS
FPGA-based design; simulation; complex systems; implementation development; performance evaluation.

1 Introduction

Field Programmable Gate Array technology [1–3] has developed considerably in recent years and commodity priced FPGA development boards now make it feasible to teach their use to students but also to deploy them in applied scientific research [4] as a source of high performance compute resource. FPGAs are finding increasing uses for application development in areas including: accelerating physics calculations [5]; agent-based modelling [6]; bio-informatics data processing [7]; image processing [8–10]; as well as cellular automata simulations [11].

Engineers have been using FPGAs for a number of years in time-critical applications, such as real-time signal processing. Recent research showing real potential for high performance computing [12], with FPGA implementations of certain problems performing better than CPU/GPU implementations [13]. This is due to inherent flexibility in custom architecture design, allowing for better mapping of some applications to the hardware it is being executed on. However, the time and expertise required for implementation development is considerably higher than for conventional software architectural development. Even though the design entry is performed using high level programming languages, the task is very different to writing a programme for a general purpose computer. We found that a hardware-oriented minds-et is required for efficient implementation and performance optimization.

Figure 1: ML605: Xilinx Virtex 6 development board

In this article we explore designing FPGA based computation engines for scientific simulations using a simple cellular automaton [14], using Conway's Game of Life [15], as an example. We used a Xilinx ML605 FPGA development board for this work. This device connects using a conventional PCIe interface slot to a PC, as is seen in Figure 1.

Our paper is structured as follows: In Section 2 we describe the general framework of use for FPGA architectures. In Section 3 we describe our use of FPGA development boards to simulate models like the Game of Life. We present some performance results in Section 4. We discuss their implications, summarise our findings and suggest some areas for further work in Section 5.

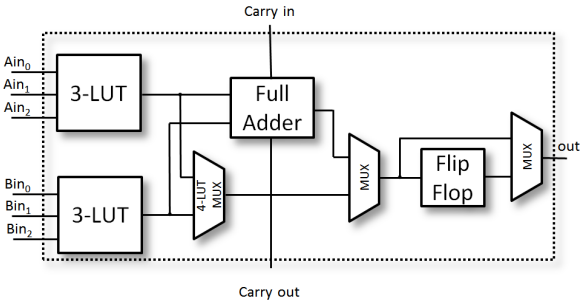


Figure 2: Simplified example of an FPGA logic slice.

2 FPGA Framework

When performing computational tasks we generally use general purpose computers. These are designed to solve any computational task, as long as we can express it as a program. This makes them versatile and relatively easy to use to solve our computational problems. However, this same ability to perform virtually any task we may ask of it limits its computational performance for any one specific task: the general purpose architecture is sub-optimal for any specific task and can be very far from optimal for some tasks.

As an alternative to using general purpose architecture, we could design a custom architecture, optimized for the specific task at hand in a way as to exploit any spatial or temporal parallelism inherent to the problem. Implementing this custom architecture on completely custom hardware through the manufacture of an application-specific integrated circuit (ASIC) would most likely result in much better performance. Unfortunately, ASIC design and manufacture is an extremely expensive and time-consuming process. Only very few applications can qualify for this approach.

There is, however, an alternative for implementing a custom architecture - reconfigurable hardware or more specifically field-programmable gate arrays. These pre-manufactured chips contain a sea of programmable logic with programmable interconnect fabric running in between. In addition, these devices also contain on-chip memory, optimized digital signal processing (DSP) blocks and a multitude of input/output pins to interconnect with other hardware, like off-chip memory. Transistor count for modern FPGAs is in the billions, allowing these devices to implement some very complex designs.

FPGA chips are designed to be reconfigurable to carry out any digital logic operation. The most basic building block of these chips is a look-up table (LUT) and register pair. 4- to 6- input LUTs are common in modern devices and can implement any boolean logic function of that many inputs. A number of LUT-Register pairs are grouped together into blocks called "logic slices". In addition to LUT-Register pairs, logic slices also generally contain dedicated arithmetic and carry logic and a number of multiplexers that can be used to combine outputs of the slice's LUTs to implement logic functions with higher number of inputs. A simplified example of a logic slice is shown in figure 2.

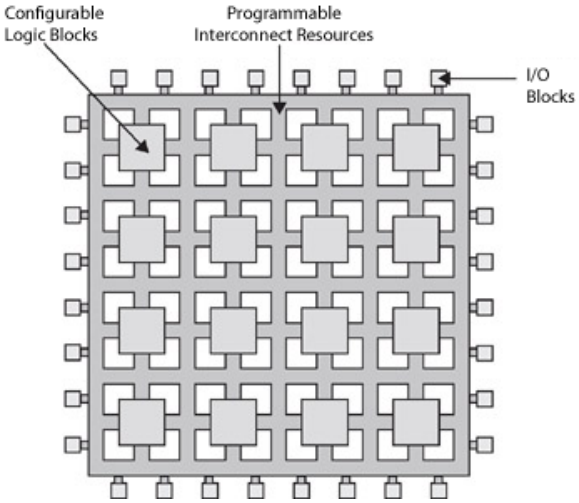


Figure 3: FPGA layout

A number of logic slices are grouped together to form configurable logic blocks (CLBs). Many thousands CLBs are located on one chip with a complex network of interconnects occupying the fabric between them (see figure 2). Just like the logic slices within CLBs, the interconnect network is programmable and can connect inputs to outputs of various slices as needed to form more complex logic functions. In addition to logic slices, FPGAs commonly contain on-chip RAM resources called block RAM (BRAM). These blocks are generally only a few kilobytes in size, but there are many of them and they can be accessed in parallel or combined into larger memories with fairly large bandwidths.

FPGA design methodology has a number of levels of abstraction ranging from the actual configuration of LUTs all the way up to high-level programming languages, such as Handle-C. It differs from software design in that an actual hardware device that would perform the desired function is being designed, rather than just a program to execute on an existing computer architecture. A flowchart of a typical design flow is shown in figure 4. The first step is architecture design, where the computational task at hand is analysed and decomposed into structural blocks with each one described in terms of its functionality and interfaces.

Once the device architecture has been designed, a hardware-description language (HDL), such as VHDL or Verilog, is used to formally describe the device. This process can utilise both the behavioural and structural description of the device and its subcomponents.

Behavioural simulation is an important next step that tests the HDL description of the problem against the de-

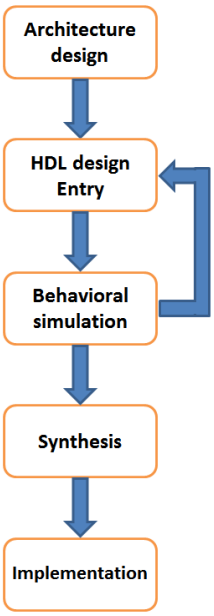


Figure 4: FPGA Design Flow

sired behaviour. At this point any behavioural differences between the HDL code and its desired behaviour are ironed out. A process called synthesis is used next to infer combinatorial and sequential logic from HDL statements. This produces an abstract representation of the desired circuit behaviour called register-transfer level (RTL) logic. This is then decomposed further into an even lower-level (logic gates and transistor level logic) representation called a netlist, generally recorded in a vendor-neutral representation using EDIF format.

The actual implementation process uses a vendor-specific tool to map the netlist representation of the designed device to the actual resources available (LUTs, BRAM, DSPs, etc) on the particular FPGA used for implementation. A process called place and route then proceeds to allocate available resources and performs routing of the signals between them as required. These are complex optimization procedures that are attempting to balance multiple trade-offs, such as chip area uptake reduction, running power consumption minimization, timing performance and implementation process runtime reduction. If one of these goals is more important than others, for example timing performance, it is possible to use a different optimization strategy to put more emphasis on increasing the maximum clock frequency. The final step is the programming file generation, which creates a bit-stream file containing the configuration settings of all the FPGA resources. This file is then simply downloaded onto the FPGA to configure it.

All of the above steps are performed using electronic design automation (EDA) tools. Implementation tools need to be vendor-specific, as they relate to the particular FPGA chip that is used for final implementation. For HDL entry and synthesis, however, 3rd party vendor-independent tools can be used (e.g. Cadence, Synplify) and the resulting netlist can be implemented using any suitable vendor. In addition to these EDA tools, designers can choose to use an even higher-level language to describe the system architecture. Recent research have been aimed at developing techniques for hardware synthesis from high level languages like C/C++ [16, 17]. Some of these are already available on the market (e.g. Handle-C, systemC); although, their uptake is slow and they are nowhere near as popular as Verilog or VHDL.

For this project we have used a Xilinx FPGA. Xilinx is one of two main vendors (Altera being the other one) of FPGA technology. ML605 development board, hosting an xc6vlx240t device from the Virtex-6 family was used. The advantage of this board is the on-board PCIe interface that can be used for exchanging the data between the host PC and the FPGA. This device contains 150,720 LUTs and 301,440 flip-flops in 37,680 logic slices and 416 36Kb block RAMs, totaling 14,976 Kb on-chip memory. We have used Xilinx ISE Design Suite 13.4 for HDL design entry, synthesis and implementation and Xilinx ISIM for behavioural simulation.

3 Simulation Formulation on FPGA

We chose to experiment with a simple Game of Life (GoL) [15] simulation for this investigation. The GoL model comprises a 2D array of bit-wise cells representing the states live or dead. At each (synchronous) time step of the model, each cell is updated according to a microscopic rule based on the number of live cells in the Moore neighbourhood of each focal cell. We can initialise the cells randomly, but that is the only randomness in the model definition, which is otherwise completely deterministic.

This class of model is interesting since GoL can be generalised to a whole family of similar automata that have exhibit emergent complex systems behaviour that can only be studied effectively using extensive simulations. FPGA technology is particularly well suited to carrying out parallel and effectively synchronous updates on a regular lattice model with localised neighbour communications. The model rules are readily implemented using primitives that can be easily implemented using FPGA component operations.

The architecture for our game of life implementation consists of two parts: the data-path and the control circuitry. The data-path is a collection of modules that store the data and operate on it. The control circuitry implements the logic required to execute the particular set of steps required to fetch the data, perform the computation and store the result. Separating the design into two parts like this makes it easier to design and maintain complex

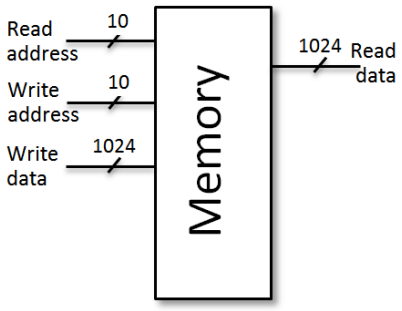


Figure 5: Custom 1024×1024 dual-port RAM.

architectures.

Along with the compute logic, the data-path must contain some sort of storage to store the state of each cell in the simulation. In software this would be stored in an array of integers located in the main memory. On the FPGA we can create a dedicated RAM to contain this data on the chip itself, using multiple blocks of BRAM. These blocks are designed to be highly configurable, allowing for a lot of flexibility and can be connected in parallel to create RAMs with any word width (number of bits per stored element) and any depth (number of stored elements). What makes BRAM even more useful is the fact that it is dual-port, meaning that we can read the state of current generation and write the computed next state at the same time.

The FPGA chip that we are using contains a total of 15Mb of storage in BRAMs, which is enough to store the state of one million cells in a 1024×1024 simulation that we want to implement (storing one bit per cell). Memory access is a significant bottleneck of modern computers. If we are going to process more than one cell at the same time, we need to have sufficient memory bandwidth to get the required data. We can bit-pack the states of a whole column of cells into a single 1024 bit-wide memory element and create a 1024-deep RAM to store each one of the columns (see figure 5). Doing this allows us to read/write the state of 1024 cells simultaneously.

To process the each cell we employ a processing element (PE) depicted in figure 6. There are 1024 instances of this element, each one computing the next generation state of the cells located in a single row, one element at a time. To avoid having to perform nine memory accesses for each computation (as the state of nine cells is required to compute this) local buffers within each processing element are employed to store the state of three consecutive cells, centered on the index of the cell, which is currently being processed. These stored states are used by the next state-computation logic and also passed to the processing elements directly above and below. The next state-computation logic is shown in figure 7. Because the value of state signals of all the neighboring cells are either one for alive or zero for dead they can be simply added together to compute the number of alive neighbors.

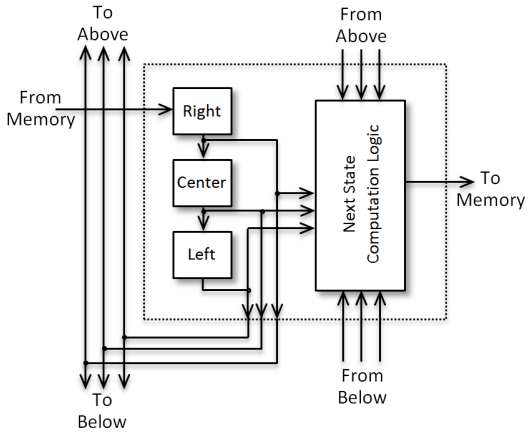


Figure 6: A single processing element contains a 3-bit shift register and combinational logic to compute the next state.

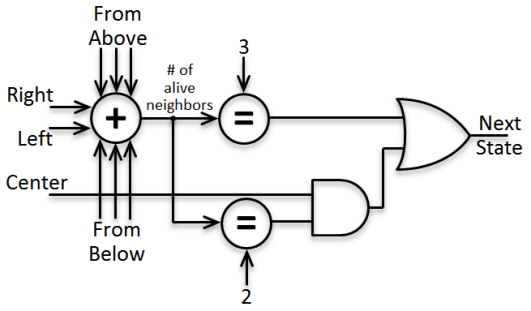


Figure 7: Combinational logic for next state computation.

To compute the next state for a column of cells three separate operations need to be performed. A memory read operation, next state computation and a memory write operation. Because next state computation requires fairly simple logic, memory read and write operations are relatively long in comparison. It is possible to pipeline these three operations to increase the overall throughput, as shown in figure 8. If the next computation logic was quite complex, it could be beneficial to also pipeline it by dividing it up into smaller computation stages; however, in this case it would not achieve any gains.

We used Verilog HDL to describe the design. Verilog description of the processing element is shown in algorithm 1. It consists of two parts: first part is the `always` statement that describes a synchronous three-element shift register using three non-blocking assignment `<=` operators. These assignments are executed simultaneously in a single clock cycle, using the value of each variable on the right hand side from the previous clock cycle. The value of this shift register is assigned to the `reg_out` port to be passed to PE above and PE below. The second part consists of computation of the number of alive neighbors by summing their states and the computation of the next state by applying the game rules. This code, along with

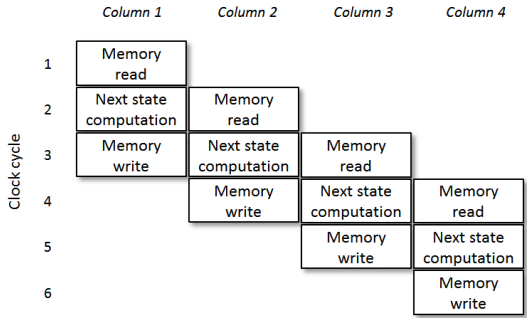


Figure 8: Pipelined memory access: reading, processing and writing each column in 3 consecutive clock cycles increases overall processing throughput.

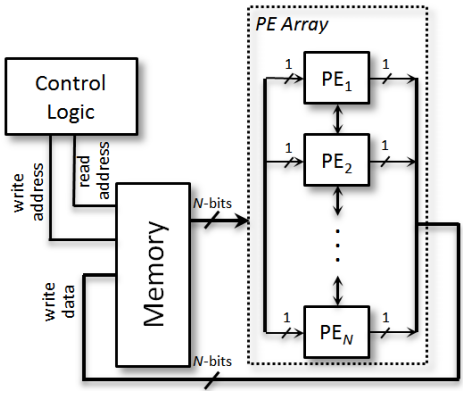


Figure 9: Processing element array contains a N PEs connected in parallel to process N rows of cells simultaneously.

the description of input and output ports, forms a Verilog module.

Each of the components in the architecture is described in a separate Verilog module, which are instantiated and interconnected using the in/out ports to complete the architecture. Modular design allows for easy reconfiguration. For example, we can change the simulation size, as long as there are enough resources on the device, to any simulation size N simply by changing the number of instances of the PE module in the PE array and BRAM configuration (figure 9). Or we can change the internal implementation of any module and as long as the external interface and functionality stays the same the design will still work.

The modules can also be parameterized, with the parameters being set at compile time. This allows for more flexibility and encourages re-usability. The processing elements, for example, can be designed to allow for q states, with q being a compile-time parameter. These can be reused in implementation of other CA, such as Game of Death [18], that require more than two states.

$N =$	512	1024	2048
Flip-flops	1550	3090	6166
LUTs	3793	7559	15113
Logic Slices	1568	3156	6571
Max clock (MHz)	269	238	187
Compilation Time (s)	149	215	375

Table 1: Resource utilization for different size simulation.

4 Performance Results

Table 1 shows resource utilization (the number of utilized flip-flops and look-up tables and also the total number of used logic slices), maximum clock frequency in megahertz at which each design is able to run and also the compilation time in seconds for three simulation sizes: 512×512, 1024×1024 and 2048×2048. The amount of utilized resources can be seen to go up linearly with the number of instantiated processing elements (512, 1024 and 2048 in each case). The maximum clock frequency goes down with simulation size. The largest simulation size, which also has the largest number of processing elements demonstrates highest throughput of 386 billion individual cells being processed per second versus 244 billion cells/s for second largest and 138 billion cells/s for the smallest. The overall performance of these three designs is 92,000 generation updates per second for the largest, 232,500 generation updates for second largest and 525,000 generation updates for the smallest.

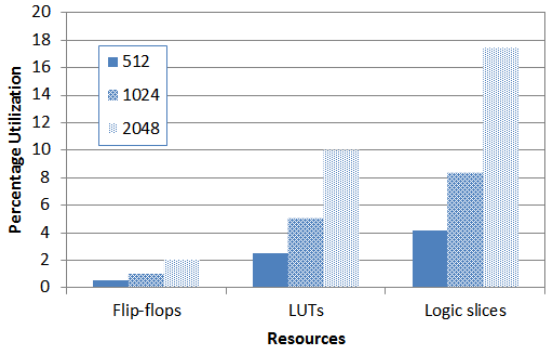


Figure 10: Resource utilization as a percentage of total available resources on xc6vlx240t device.

Figure 10 shows resource utilization figures as a percentage of the total resources available on xc6vlx240t device. It can be seen that even for a simulation as large as 2048×2048 elements, logic slice utilization is under 18%.

5 Discussion & Conclusions

The development process of this design followed the design flow shown in figure 4. The overall time was roughly split half and half between HDL entry/behavioural simulation and architecture design/optimization.

Algorithm 1 Verilog code for the Processing Element.

```
1  always @(posedge CLK)
2  begin
3          right <= Data_in;
4          centre <= right;
5          left <= centre;
6  end
7  assign reg_out = {left,centre,right};
8  assign numAlive = (fromAbove_in[0] + fromAbove_in[1] + fromAbove_in[2] + left + right
9                  + fromBelow_in[0] + fromBelow_in[1] + fromBelow_in[2]);
10 assign nextGen = (numAlive == 4'd3) || ((numAlive == 4'd2) && centre);
```

We found both of these tasks to be highly iterative and the use of good EDA tools can make a big difference to the amount of time/effort taken up by these tasks.

Verilog HDL is a high-level language; however, we found writing Verilog code to be very different to writing a program in a software language, partly because the Verilog 'program' actually represented a piece of hardware with certain functionality, rather than an algorithm normally described by a program in a software language. The inherent notion of time in the process of describing synchronous logic also made it difficult to get the HDL description right the first time. This is where the behavioural simulation proved to be extremely valuable.

Performance of compute engines such as the one described here can be measured using throughput, a product of how much computation is performed in one clock cycle and the clock frequency. To increase performance, we can either utilize a large number of the processing elements or optimize the processing elements to increase the clock frequency, or both. We have tried a number of ways to simplify/optimize the processing element module and the processing element array, for example by replacing the series of 4-bit additions on lines 8 and 9 in algorithm 1 with an adder tree composed of smaller 2- and 3-bit adders and one 4-bit adder, as illustrated in algorithm 2.

These attempts resulted in simpler RTL representation of the design. This, however, did not necessarily equate to any significant gains in maximum clock frequency or resource utilization. This is partly due to the synthesis tools running various automated logical and physical optimization of the RTL and doing a fairly good job for a relatively simple design such as this one and partly due to the nature of the way digital logic is implemented on FPGAs - using a series of look-up tables.

We found it difficult to manually optimize the design prior to running the synthesis, as it was hard to tell whether what we considered simpler and faster would result in simpler and faster final implementation. This process of manual optimization turned into a series of trial and error steps to find out what worked and what didn't. Unfortunately, with a total compilation time of over 6 minutes for the 2048×2048 design this has become a very time consuming process.

Algorithm 2 Verilog code representing an adder tree.

```
1  // Allocating 2-bit temporary storage:
2  wire [1:0] temp1,temp2, temp3;
3  wire [2:0] temp4; // 3-bit temporary storage
4  assign temp1 = fromAbove_i[0] + fromAbove_i[1]
5           + fromAbove_i[2];       // 2-bit adder
6  assign temp2 = fromBelow_i[0] + fromBelow_i[1]
7           + fromBelow_i[2];       // 2-bit adder
8  assign temp3 = left + right;     // 2-bit adder
9  assign temp4 = temp1+temp2;      // 3-bit adder
10 assign numAlive = temp4 + temp3;// 4-bit adder
```

The numbers in table 1 indicate that this design scales fairly well with simulation size in terms of resource utilization. Large designs, however, cannot be clocked at the same frequency as the smaller one, even though the main difference between them is the number of instantiated processing elements (all having the same complexity). This is likely to be due to more complex internal signal routing that introduces more delay as resource utilization increases.

Introducing more processing elements, whether it is for a larger simulation size or to increase the throughput even further by processing more than one column at a time, would most likely decrease the maximum clock frequency even further. Introducing more processing elements (if the memory bandwidth allows) would obviously increase throughput, but only up to a certain point, since this increase will have a detrimental effect on the maximum clock frequency.

Figure 10 shows that there is still a large portion of unused resources left on the device. This can either be utilized by more PEs to process multiple columns at the same time as stated above, or it can also be utilized to compute any required measurements, such as statistical metrics, in real-time as the simulation progresses, thus saving time on further analysis. Performing the required measurements in parallel to the running of the simulation can also greatly reduce the amount of data that needs to be taken from the

device, by discarding the raw data and only reading the required metrics.

In summary, FPGA technology has a promising outlook [19] and we have found it to be well suited to this sort of complex systems simulation, that makes use of regular data structures, localised communications and good use of component primitive operations. FPGA implementations of this class of model will potentially aid the systematic exploration of a model space by providing very fast but cheap simulation platforms.

There is additional scope to develop some of the model measurement algorithms (such as tracking density of live cells) and mechanisms for collecting statistics that would even further enhance this platform's suitability.

References

[1] Oldfield, J.V., Dorf, R.C.: Field programmable gate arrays - Reconfigurable logic for rapid prototyping and implementation of digital systems. Number ISBN 0-471-55665-3. Wiley (1995)

[2] Chu, P.P.: FPGA Prototyping by VERILOG Examples. Number ISBN 978-0-470-18532-2. Wiley (2008)

[3] Herbordt, M.C., Gu, Y., VanCourt, T., Model, J., Sukhwani, B., Chiu, M.: Computing Models for FPGA-Based Accelerators. Computing in Science & Engineering **10**(6) (November/December 2008) 35–45

[4] Stitt, G., george, A., Lam, H., Readdon, C., Smith, M., Holland, B., Aggarwal, V., Wang, G., Coole, J., Koehler, S.: An End-to-End Tool Flow for FPGA-Accelerated Scientific Computing. IEEE Design and Test of Computers **July/August** (2011) 68–77

[5] Danese, G., Leporati, F., Bera, M., Giachero, M., Nazzicari, N., Spelgatti, A.: An accelerator for physics simulations. Computing in Science and Engineering **9**(5) (September 2007) 16–25

[6] Chen, E., Lesau, V.G., Sabaz, D., Shannon, L., Gruver, W.A.: Fpga framework for agent systems using dynamic partial reconfiguration. In: Proc. 5th Int. Conf on Industrial Applications of Holonic and Multi-Agent Systems (HoloMAS). Number 6867 in LNAI, Toulouse, rance (29-31 August 2011) 94–102

[7] Anan'ko, A.G., Lysakov, K., Shadrin, M.Y., Lavrentiev, M.M.: Development and application of an fpga based special processor for solving bioinformatics problems. Pattern Recognition and Image Analysis **21**(3) (2011) 370–372

[8] Gribbon, K.T., Bailey, D.G., Bainbridge-Smith, A.: Development issues in using fpgas for image processing. In: Development Issues in Using FPGAs for Image Processing?, Proceedings of Image and Vision Computing New Zealand 2007, Hamilton, New Zealand (2007) 217–222

[9] Bailey, D.: Design for Embedded Image Processing on FPGAs. Wiley (2011) ISBN 9780470828496.

[10] Huang, Q., Wang, Y., Chang, S.: High-performance fpga implementation of discrete wavelet transform for image processing. In: Proc. 2011 Symposium on Photonics and Optoelectronics (SOPO), Wuhan (16-18 May 2011 2011) 1–4

[11] Olson, A.: Towards fpga hardware in the loop for qca simulation. Master's thesis, Rochester Institute of Technology, New York (May 2011)

[12] El-Ghazawi, T., El-Araby, E., Huang, M., Gaj, K., Kindratenko, V., Buell, D.: The promise of high-performance reconfigurable computing. Computer **41**(2) (feb. 2008) 69 –76

[13] Che, S., Li, J., Sheaffer, J., Skadron, K., Lach, J.: Accelerating compute-intensive applications with gpus and fpgas. In: Application Specific Processors, 2008. SASP 2008. Symposium on. (june 2008) 101 –107

[14] Murtaza, S., Hoekstra, A.G., Sloot, P.M.A.: Cellular automata simulations on a fpga cluster. High Performance Computing Applications **25, Online**(2) (October 2010) 193–204

[15] Gardner, M.: Mathematical Games: The fantastic combinations of John Conway's new solitaire game "Life". Scientific American **223** (October 1970) 120–123

[16] Edwards, S.A.: The challenges of hardware synthesis from c-like languages. In: Proceedings of the 2005 Design, Automation and Test in Europe. (March 2005) 66 –67

[17] Edwards, S.: The challenges of synthesizing hardware from c-like languages. Design & Test of Computers, IEEE **23**(5) (May 2006) 375 –386

[18] Hawick, K., Scogings, C.: Cycles, transients, and complexity in the game of death spatial automaton. In: Proc. International Conference on Scientific Computing (CSC'11). Number CSC4040, Las Vegas, USA (July 2011)

[19] Constantinides, G.A., Nicolici, N.: Surveying the landscape of fpga accelerator research. In: IEEE Design and Test of Computers. Volume July/August. (2011) 6–7

Parallel Counter Formulation using a Generator Polynomial Expansion

Lee A. Belfore II
Department of Electrical and Computer Engineering
Old Dominion University
Norfolk, VA 23323
USA

Abstract— *Parallel counters have been studied for several decades as a component in fast multipliers and multi-operand adder circuits. Efficient design of these functional units can produce fast & efficient multipliers and signal processors. Proposed in this paper is a GF(2) generator polynomial expansion that can be used to specify parallel counters. The mathematical formalism provides a general way of describing parallel counters. Furthermore, the expansion suggests a gate level implementation for parallel counters.*

Keywords: Parallel counter, generator polynomial, synthesis.

1. Introduction

Parallel counters are combinational logic circuits that take as input a number of equally weighted binary inputs and outputs a binary number which is the number of inputs inputs that are one. Parallel counters are used in applications which are sped up when multiple addends can be concurrently added to form a result. This occurs in such applications as fast combinational multipliers and signal processing applications.

Because applications vary thereby changing the number of operands, a generalized approach for synthesizing parallel counters is also of value. Many of the approaches build general parallel counters from (3,2) parallel counters (e.g. full adders) as well as higher order parallel counter building blocks. Indeed, references [1], [2] provide good overviews of the design and specification of parallel counters. Applications of parallel counters in fast multipliers were first formulated in [3]. Because parallel counters are used in high performance applications, they are often designed with high speed and compactness in mind to manage the complementary issues of performance and implementation complexity.

This paper is organized into six sections including an introduction, a presentation of the generator polynomial formulation, an analysis of power of two combinatorics as applied to the proposed GF(2) generator polynomial, an analysis of the properties of generator polynomial coefficients, a proposed circuit synthesis approach, and a summary.

2. Generator Polynomial Formulation

Before introducing the generator formulation, we first state some assumptions. First, the GF(2) algebra is used in the formulation. Recall that GF(2) has two operators, *Exclusive-OR* and *AND*, which will be denoted by $+$ and $\cdot$ (often implied) respectively. Literals in the GF(2) system nominally take on the values 0 and 1. The GF(2) system serves as the mathematical system that is used in a variety of applications including, for example, encryption and error correcting codes. In this section, we explore a generator polynomial formulation for parallel counters.

Rather than formulating fixed coefficient generator polynomial, the generator polynomial coefficients described here are determined by literals. A literal a can be represented by the following factor

$$\mathcal{F}(r) = 1 + a \cdot r, \tag{1}$$

where r is the indeterminate for the generator polynomial. Representing literals as in (1) suggests a mathematical formulation for combining literal factors into higher order generator polynomials. For example, the product of the factors for a_1, a_2, and a_3 is

$$\begin{aligned} \mathcal{F}^{(3)}(r) &= (1 + a_1 r)(1 + a_2 r)(1 + a_3 r) \\ &= 1 + (a_1 + a_2 + a_3)r \\ &\quad + (a_1 a_2 + a_1 a_3 + a_2 a_3)r^2 \\ &\quad + (a_1 a_2 a_3)r^3 \\ &= 1 + F_1^{(3)} r + F_2^{(3)} r^2 + F_3^{(3)} r^3. \end{aligned} \tag{2}$$

Note that the degree for each coefficient matches the number of literals in the contributing product terms, or cubes, included in the coefficient. Furthermore, and curiously, the second and first degree coefficients, respectively, are the sum and carry functions for the full adder function for the literals a_1, a_2, and a_3. Indeed, these also define the encoded outputs for the (3,2) parallel counter:

$$\begin{aligned} F_2^{(3)} = S_1 = \quad \text{Carry} \quad &= a_1 a_2 + a_1 a_3 + a_2 a_3 \\ F_1^{(3)} = S_0 = \quad \text{Sum} \quad &= a_1 + a_2 + a_3. \end{aligned} \tag{3}$$

Traditionally, the full adder carry function is expressed as sum of products form with AND and OR primitives. While in GF(2) '+' is exclusive-OR, it is easy to confirm the correctness of the full adder carry function. Another interesting observation for this simple case is that each degree i coefficient is the sum of all distinct cubes that are the product of a different combination of i literals.

Extending this result, a generator polynomial can be formulated in the general case for n literals $a_1, a_2, ..., a_n$

$$\begin{aligned} \mathcal{F}^{(n)}(r) &= \prod_{i=1}^{n}(1+a_i r) \\ &= 1 + F_1^{(n)} r + F_2^{(n)} r^2 + \cdots + F_n^{(n)} r^n \\ &= 1 + \sum_{d=1}^{n} F_d^{(n)} r^d. \end{aligned} \quad (4)$$

From (4), several useful properties can be elicited.

Theorem 1: Each coefficient $F_d^{(n)}$ in $\mathcal{F}^{(n)}(r)$ is the exclusive-OR of all $\binom{n}{d}$ distinct degree d cubes.
Proof: From the binomial expansion,

$$\mathcal{F}(r) = (x_0 + x_1)^n, \quad (5)$$

which can also be expressed as

$$\mathcal{F}(r) = \prod_{i=1}^{n}(x_0 + x_1). \quad (6)$$

In (6), the product is composed of 2^n terms where each can be uniquely identified by an n bit codeword representing the contribution of either x_1 or x_0 from each of the n factors at position i. Furthermore, each individual product term codeword can uniquely be defined by

$$\mathcal{S}_m = \Big\|_{i=1}^{n} \begin{cases} x_1 & \text{for} \quad m_{i-1} = 1 \\ x_0 & \text{for} \quad m_{i-1} = 0 \end{cases} \quad (7)$$

where $\|$ is the concatenation operator and m be the "value" of this codeword. The codeword value is determined by associating a 1 for x_1, and 0 for x_0 in the following expression

$$m = \sum_{i=1}^{n} \begin{cases} 2^{i-1} & \text{for} \quad x_1 \\ 0 & \text{for} \quad x_0. \end{cases} \quad (8)$$

Generalizing (6), we substitute $x_0 = 1$ and $x_1 = a_i r$ resulting in the unexpanded and expanded polynomial forms given in (4). Since it has been shown that all codewords, and therefore all terms, are generated by the binomial product, (4) is exhaustive and includes all unique cubes. In this formulation, each polynomial coefficient i will be the sum of all different cubes composed from exactly i literals, because cubes having greater or fewer literals will be included in higher or lower degree coefficients. □

Lemma 1: The number of degree d cubes in (4) is $N_d = \binom{n}{d}$.

Proof: The Binomial Theorem can be used to determine the number of product terms fitting a particular composition, i.e. the multiplicity of x_1 and x_0. Consider the expression

$$\mathcal{F}(r) = \prod_{i=1}^{n}(1 + a_i r). \quad (9)$$

In (9), 2^n cubes result, and each can be associated with an encoding, m, that reflects the contribution of the literals, a_i. Each cube is unique because it reflects of a different selection of literals from each of the respective n factors. Furthermore, each cube's degree, d, is the number of literals in the product and is also the the number of ones in the encoding m. Thus, the number of cubes for a particular degree follows from the binomial coefficient

$$N_d = \binom{n}{d}, \quad (10)$$

where N_d is the number of degree d cubes. Each degree d coefficient, $F_d^{(n)}$, is the Exclusive-OR of the N_d different degree d cubes. □

3. Power of Two Combinatorics

In §2, the first and second degree coefficients from $\mathcal{F}^{(3)}(r)$ were the sum and carry functions for the full adder. In order to generalize the formulation to parallel counters of arbitrary size, the number of cubes contributing to a coefficient will be determined to study coefficient properties. Since the polynomial is GF(2), the focus will be determining whether the number of cubes contributing to specific coefficients is even or odd.

Lemma 2: For an integer $k > 1$, $2^{(2^k-1)}$ is largest power of two by which $(2^k)!$ is divisible.[1]
Proof: The lemma can be proven by repeatedly factoring out two. Note that the number of even terms in $(2^k)!$ is $\frac{2^k}{2}$, the number divisible by four are $\frac{2^k}{4}$, the number divisible by 2^i for $i \leq k$ are $\frac{2^k}{2^i}$. Let p be the number of twos that can be factored out and is expressed as

$$\begin{aligned} p &= \sum_{i=1}^{k} \frac{2^k}{(2^i)} \\ &= \sum_{i=1}^{k} 2^{k-i} \\ &= \sum_{j=0}^{k-1} 2^j \\ &= 2^k - 1. \end{aligned} \quad (11)$$

Equation (11) accounts for all factors of two and the final step in (11) establishes that $(2^k)!$ is divisible by $2^p = 2^{(2^k-1)}$. □

Lemma 3: For $k > j$, $(2^k - 2^j)!$ has exactly $(2^k - 2^j - (k-j))$ factors of two.

[1]This property is generally attributed to Legendre.

Proof: By Lemma 2, $(2^k)!$ is divisible by two with multiplicity $(2^k)-1$. Note that $(2^k-2^j)!$ has 2^j fewer factors and will thus reduce the multiplicity. Noting further that $(2^k-2^j)!$ is on a 2^j boundary, the multiplicity is reduced by $(2^j)-1$. The very last factor, 2^k has only been reduced by 2^j, and so multiplicity must be further reduced by $(k-j)$. We can express the number of two factors in $(2^k-2^j)!$ as

$$\begin{aligned} p_2 &= (2^k-1)-(2^j-1)-(k-j) \\ &= 2^k-2^j-(k-j). \end{aligned} \tag{12}$$

□

Lemma 4: For $k>j$, $\binom{2^k}{2^j}$ is divisible by 2^{k-j}.

Proof: Because

$$\binom{2^k}{2^j} = \frac{(2^k)!}{(2^k-2^j)!(2^j)!} \tag{13}$$

and using the previous Lemmas, the divisibility by powers of two can be expressed as

$$\begin{aligned} p_2 &= 2^k-1-((2^k-1)-(2^j-1)-(k-j))- \\ &\quad (2^j-1) \\ &= (k-j). \end{aligned} \tag{14}$$

Thus, $\binom{2^k}{2^j}$ is divisible by 2^{k-j}. □

Lemma 5: $\binom{2^k+i}{2^j}$ is even for $k>j$ and $i<2^j$.

Proof: For $i=0$, $\binom{2^k}{2^j}$ can easily be shown to be even by applying Lemma 4. The remaining cases can be examined by rearranging the factors in the numerator and denominator as follows:

$$\begin{aligned} \tbinom{2^k+i}{2^j} &= \tfrac{(2^k+i)!}{(2^k+i-2^j)!(2^j)!} \\ &= \left(\tfrac{(2^k+i)\cdots(2^k+1)}{(2^k-2^j+i)\cdots(2^k-2^j+1)}\right)\left(\tfrac{(2^k)!}{(2^k-2^j)!(2^j)!}\right) \\ &= \tfrac{(2^k+i)\cdots(2^k+1)}{(2^k-2^j+i)\cdots(2^k-2^j+1)}\tbinom{2^k}{2^j}. \end{aligned} \tag{15}$$

Because the numerator of $\frac{(2^k+i)\cdots(2^k+1)}{(2^k-2^j+i)\cdots(2^k-2^j+1)}$ starts on a 2^k boundary and the denominator starts on a 2^j boundary not divisible by 2^{j+1}, each is divisible by the same number of factors of two and thus is odd. By Theorem 2, the second term, $\binom{2^k}{2^j}$ has exactly $k-j$ factors of two. Thus, $\binom{2^k+i}{2^j}$ is even for $k>j$ and $i<2^j$. □

Lemma 6: $\binom{2^k+i}{2^k}$ is odd for $i<2^k$.

Proof: Note that

$$\begin{aligned} \tbinom{2^k+i}{2^k} &= \tfrac{(2^k+i)!}{(2^k)!(i)!} \\ &= \tfrac{((2^k+i)\cdots(2^k+1))(2^k)!}{(2^k)!(i)!} \\ &= \tfrac{(2^k+i)\cdots(2^k+1)}{(i)!}. \end{aligned} \tag{16}$$

Since $((2^k+i)\cdots(2^k+1))$ and $(i)!$ begin on 2^k boundaries, each is divisible by two in precisely the same way and $\binom{2^k+i}{2^k}$ is thus odd. □

4. Properties of Generator Polynomial Coefficients

In this section, the properties of generator polynomial coefficients will be studied in terms of their relation to the number of literals that are one. In the previous section, a collection of combinatorial relations were developed to ascertain whether a particular combination specification was even or odd. This is relevant to studying of coefficients because if we can relate the number of literals that are one to cubes in polynomial coefficients, coefficient values as well as their relationships.

We will begin with the following definitions.

Definition 1: An instance, ι, of a degree d cube is denoted by C_d^ι and is an instance ι of a product term composed of d different literals.

Definition 2: A cube of degree d, C_d^ι **covers** *a cube C_e^s, $e \le d$ when the literals that form C_e^s are a subset of the literals used to form to C_d^ι.*

Definition 3: The quantity $|\mathbf{A}|$ is the number of literals in the vector $\mathbf{A}=a_1a_2\cdots a_n$ that are one.

From these definitions, we can state the following two lemmas.

Lemma 7: If the cube $C_{2^k}^\iota = 1$ and $j<k$, then each covered cube $C_{2^j}^\zeta = 1$.

Proof: If $C_{2^k}^\iota = 1$, then all of the literals in the cube must be one. Furthermore, any cube ζ formed by a subset of these literals must also be one so $C_{2^j}^\zeta = 1$. □

Lemma 8: The cube $C_{2^k}^\iota$ covers $\binom{2^k}{2^j}$ degree 2^j cubes when the 2^j literals are subset of the 2^k literals used to form $C_{2^k}^\iota$.

Proof: Since literals in the degree 2^j are a subset of the literals formed from $C_{2^k}^\iota$, the total number of different cubes is the different 2^j degree cubes selected from a pool of 2^k, or $\binom{2^k}{2^j}$ □

Next, the following theorem considers the case where $|\mathbf{A}|=2^k$ and the impact on the degree 2^j, $j\le k$ coefficients $F_{2^j}^{(n)}$.

Theorem 2: If $|\mathbf{A}|=2^k$, exactly one cube $F_{2^k}^{(n)}$ evaluates to one and results in $F_{2^k}^{(n)}=1$. All other coefficients, $F_{2^j}^{(n)}=0$ for $j<k$.

Proof: If $|\mathbf{A}|=2^k$, then exactly one the cube, $C_{2^k}^\iota$ is one and it is included in $F_{2^k}^{(n)}$, so therefore $F_{2^k}^{(n)}=1$.

For $j<k$, the coefficient $F_{2^j}^{(n)}$ includes the contribution of $\binom{2^n}{2^j}$ cubes, all covered by $C_{2^k}^\iota$, or $\binom{2^k}{2^j}$ are one. By Lemma 4, $\binom{2^k}{2^j}$ is divisible by 2^{k-j} and thus is even. Since $\binom{2^k}{2^j}$ is even, $F_{2^j}^{(n)}=0$. □

One interpretation of Theorem 2 is that if $|\mathbf{A}|=2^k$, all smaller cubes $C_{2^j}^\zeta$ that are formed from the same literals are covered and effectively masked out because the total number of these smaller cubes is even. Next, we generalize this result for cases where $|\mathbf{A}|\neq 2^k$ in the following two corollaries.

Corollary 1: For $0 \leq i < 2^k$, if $|\mathbf{A}| = (2^k + i)$, then $F^{(n)}_{2^k} = 1$.

Proof: Since (2^k+i) ones cover $\binom{2^k+i}{2^k}$ degree 2^k cubes and each of these 2^k cubes is one, then the number of degree 2^k cubes are odd by Lemma 6, and $F^{(n)}_{2^k} = 1$. □

Corollary 2: If $|\mathbf{A}| = (2^k, 2^{k+1})$, one cube $C^{\iota}_{2^k}$ covers 2^k literals and all cubes composed of these literals. The remaining $|\mathbf{A}| - 2^k$ literals are disjoint from $C^{\iota}_{2^k}$ and uncovered.

Proof: By Corollary 1, an odd number of cubes are one in the coefficient $F^{(n)}_{2^k}$. Without loss of generality, we select one representative cube $C^{\iota}_{2^k}$ and the remaining even number of cubes effectively cancel one and other out. By Theorem 2, all lower order cubes covered by $C^{\iota}_{2^k}$ make no contribution to their respective lower order generator polynomial coefficients. The remaining $|\mathbf{A}| - 2^k$ uncovered literals, $\mathbf{A^d}$, are disjoint from the representative 2^k literals. □

Corollary 3: The disjoint literals from Corollary 2, $\mathbf{A^d}$, form a lower order generator polynomial representative of its constituent disjoint literals.

Proof: The original general polynomial was given in (4). Factoring out the contribution of the 2^k literals that were one leaves a generator polynomial with $|\mathbf{A^d}| = |\mathbf{A}| - \mathbf{2^k}$ factors. □

We can use Corollaries 2 and 3 as a sieve to associate literals that are one with power of two degree generator polynomial coefficients. At this point, we are ready to present the main result of this paper in the following theorem.

Definition 4: k_m is the largest power of two such that 2^{k_m} does not exceed n or $2^{k_m} \leq n < 2^{(k_m-1)}$.

Theorem 3: For a given assignment of the literals $a_1 a_2 \cdots a_n$, the coefficients from $F^{(n)}_{2^k}$ for $k \in \{k_m, ..., 0\}$ encode a binary number which is the number literals that are one and is expressed as

$$|\mathbf{A}| = \sum_{k=0}^{k_m} F^{(n)}_{2^k} 2^k. \tag{17}$$

Proof: Given $0 \leq |\mathbf{A}| \leq n$, where $n = 2^{(k_m+1)} - 1$. In the trivial case where $|\mathbf{A}| = 0$, all literals are zero and therefore all $F^{(n)}_{2^k} = 0$.

Next, we will assume that $|\mathbf{A}| = 2^k$. In this case, the k^{th} bit of the encoding is one and the rest are zero by Theorem 2 and $F^{(n)}_{2^k} = 1$.

The general case can be shown by repeated application of Corollaries 2 and 3. If $|\mathbf{A}| = (2^k, 2^{k+1})$, for higher contributions, $[k+1, k_m]$ are zero because the cubes for the respective generator polynomial coefficients must be zero. By Corollary 2, $F^{(n)}_{2^k} = 1$ and it covers the included 2^k literals in the lower order generator coefficients and contributes 2^k to the literal count. By excluding the covered cubes by Corollary 2 and specifying a reduced order generator polynomial of the remaining literals by Corollary 3, the process is repeated until all literals are covered, resulting in the literal count specified by (17). □

From Theorem 3 it is useful to define the binary number that gives the literal count as follows.

Definition 5: Concatenating the coefficients from $F^{(n)}_{2^k}$ for $k \in \{k_m, ..., 0\}$ results in the binary string

$$\mathcal{S} = F^{(n)}_{2^{k_m}} \parallel F^{(n)}_{2^{k_m-1}} \parallel \cdots \parallel F^{(n)}_{2^0}. \tag{18}$$

5. Parallel Counter Synthesis

To this point, we have developed a novel mathematical framework for describing parallel counters in the general case. In this section, we address some basic implementation issues. Because the generator polynomial is formed with AND and Exclusive-OR operations, in the context of the generator polynomial coefficients, we can specify AND Exclusive-OR logic circuits to implement the respective logic functions.

Theorem 4: The expressions that specify $\mathcal{S}$ can be implemented with combinational logic circuits.

Proof: This is shown by implementing the logic functions defined in Theorem 3 with Exclusive-OR gates and two-input AND gates.

Assume that we start with the generator polynomial $\mathcal{F}^{(j-1)}$. We can express $\mathcal{F}^{(j)}$ as

$$\begin{aligned} \mathcal{F}^{(j)} &= \mathcal{F}^{(j-1)}(1 + a_j r) \\ &= \mathcal{F}^{(j-1)} + a_j \mathcal{F}^{(j-1)} r. \end{aligned} \tag{19}$$

Equation (19) implies three different relations describing the determination of the coefficients in $\mathcal{F}^{(j)}$ depending on the degree of the associated coefficient. For the first degree coefficient,

$$F^{(j)}_1 = F^{(j-1)}_1 + a_j. \tag{20}$$

Indeed, in the general case, the first degree coefficient is the Exclusive-OR of all literals. For the highest degree, degree j case,

$$F^{(j)}_j = F^{(j-1)}_{j-1} a_j. \tag{21}$$

In this case, the highest degree coefficient is the AND of all literals. In the remaining cases, the coefficients are

$$F^{(j)}_k = F^{(j-1)}_k + a_j F^{(j-1)}_{k-1}, \tag{22}$$

where $k < j$. Equation (22) indicates a primitive unit consisting of an AND gate and an Exclusive-OR gate can be used to construct arbitrary sized parallel counters.

Equations (20)-(22) describe how the j^{th} degree generator polynomial can be determined from the $(j-1)^{\text{th}}$ generator polynomial and literal a_j. Furthermore, each additional literal contributes a layer to a combinational logic circuit to give the implementation for the next higher order polynomial. Beginning with the first degree generator polynomial,

we can construct the circuit for the n^{th} degree generator polynomial. The resulting parallel counter outputs are the are simply the coefficients from the generator polynomial identified in Definition 5. □

Theorem 4 can be used to describe all the circuitry necessary to build the generator polynomial coefficients in the general case. Indeed, the actual circuit can be pruned by making a couple of observations. First, given $2^{k_m} \leq n < 2^{k_m-1}$, only the coefficients up to 2^{k_m} need be considered. In addition, coefficients not contributing to coefficients identified in $\mathcal{S}$ may be pruned as well.

The mathematical formulation just presented forms the basis for the synthesis of parallel counters. Because the mathematics employs AND and Exclusive-OR operations for the expressions, these can be mapped directly to the implementation. Figure 1 gives a gate level schematic for a (7,3) parallel counter. In addition, this figure is labeled to show the generator polynomial coefficients used to synthesize the circuit. Parts of the circuit that do not contribute to the count are highlighted in gray and can be pruned from the circuit representing the generator polynomial.

6. Conclusion

In this paper, we present a novel formalism to describe parallel adders using a GF(2) generator polynomial expansion. We have proven several useful properties that result from studying the nature of the polynomial coefficients. In particular, by selecting the coefficients that are associated with a degree that is a power of two, the parallel counter encoding for the literals results. Since the GF(2) system is based on AND and Exclusive-OR operators, we can further specify and synthesize parallel counter circuits of any order. Future efforts will include analysis and research into optimized parallel counter circuits.

References

[1] E. E. Swartzlander, Jr, "Parallel counters," *IEEE Transactions on Computers*, vol. C-22, no. 11, pp. 1021–1024, November 1973.

[2] ——, "A review of large parallel counter designs," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI Emerging Trends in VLSI Systems Design (ISVLSI'04)*, 2004.

[3] L. Dadda, "Some schemes for parallel multipliers," *Acta Frequenza*, vol. 45, pp. 574–580, 1965.

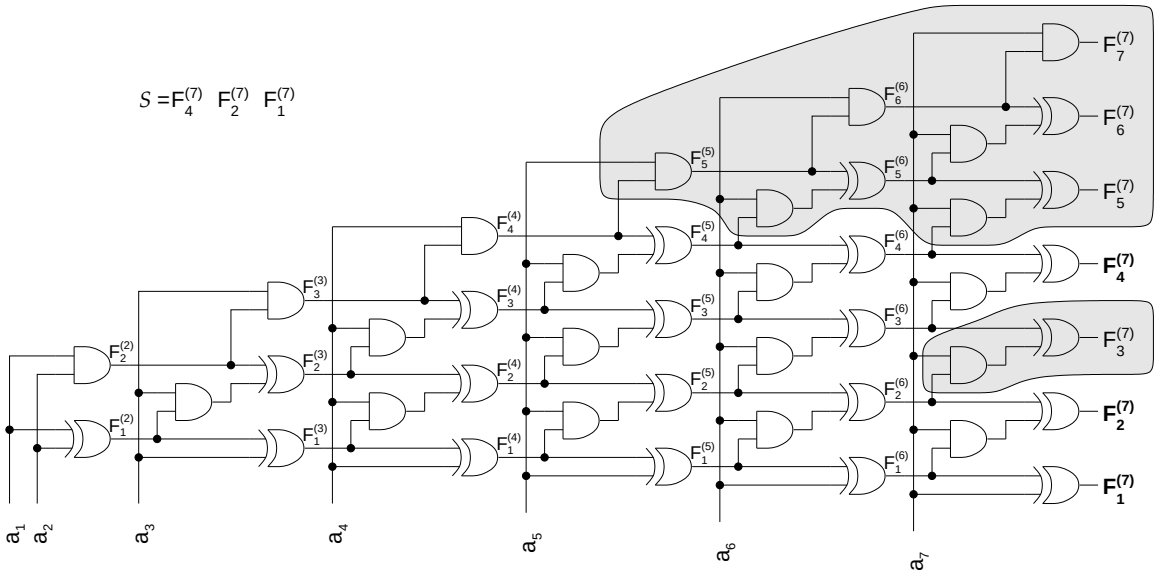


Fig. 1

(7,3) PARALLEL COUNTER. NOTE THE PRUNED COMPONENTS (GRAY) AND THE GENERATOR POLYNOMIAL COEFFICIENTS.

Performance Bound Energy Efficient L2 Cache Organization for Emerging Workload for Multi-Core Processor: A Comparison of Private and Shared Cache

Ramya Arun and Eugene John
Department of Electrical and Computer Engineering
University of Texas at San Antonio,
One UTSA Circle, San Antonio, TX 78249, U. S. A.

Abstract *- While multi-core design brings opportunity for more performance and power efficiency, there are performance challenges as the number of cores increase. This increase in performance can only be harnessed when issues like memory speed bottleneck and decreasing average cache size per core are overcome. The target applications must be known to maximize the performance improvements through increasing parallelism and cache hit rates. Also applications require highly diverse cache configurations for optimal energy consumption in the memory hierarchy to support their implementation. In this paper various cache organizations for multi-core processors are simulated and their performance and energy tradeoff are studied for emerging workloads. Finally, the trend in performance and energy consumption for the optimized cache configurations with increasing number of cores is analyzed.*

Keywords: *L2 cache, PARSEC benchmark, Multi-core, Energy efficient cache, Workload performance optimization*

1 Introduction

The challenge of every microprocessor designer is to improve the processor performance. The quest for better performance gives rise to various challenges. The processor speed is increasing due to advancement semiconductor technology and due to instruction level parallelism, among many other factors. These are multiplicative and increases the speed of processor tremendously. But the performance improvement is not only dependent on the speed of the processor but also greatly dependent on the speed at which memory components can supply instructions and data. Modern microprocessors get faster at an average rate of 60% per year while system memory speed increases by only 7% per year [1]. This performance gap can to a certain extend be bridged by good cache design.

In the past decade the performance of a processor was synonymous with clock frequency. Thus the quest to increase operating frequency has increased the power consumption of the processor. Power consumption increases linearly with frequency. Memory can consume as much as 50% of the system power in a microprocessor. Thus memory organization is not only a performance bottleneck, but also a major factor in efficient power design.

2 Background and Motivation

Memory bandwidth can be increased by implementing multiple main memory ports and using large number of memory banks. Thus memory bandwidth bottleneck is not a main memory problem but is an issue due to interconnect. The problem lies in how the memory units get interfaced to the cores.

The main performance challenge of increasing number of cores is the network that connects the various cores to each other and the network connecting core and main memory. Multi-core systems rely on buses or rings for interconnects. Increasing the number of cores demand data at a higher rate. But interconnects don't scale and hence becoming a bottleneck. This slows overall system performance. Memory bandwidth problem can be mitigated by distributing caches along with the cores [2].

The private cache will have smaller hit access time and smaller energy per access. Hits to the private partitions are fast, while hits to neighboring partitions are slower. If the threads running in different core have independent data needs then this type of partition will evolve as the best configuration. Dedicated cache provides rapid access for each core as it is good for threads with strong locality and increases memory bandwidth. As number of cores increases the average cache size per core will decrease. Number of cache blocks per core will be reduced which causes additional capacity and conflict misses. Potential solutions for this problem are to increase the number of cache blocks by increasing cache sizes and reducing cache block size and by increasing the level of associativity. Limitations of these solutions are: larger the cache size larger the power dissipation and chip area. Larger also means slower, sometimes limiting the clock rates of the cores.

As the transistors are scales down in size architects can integrate more number of cores in the same area which was occupied by a single core of older technologies. As the number of cores increases the available cache space has to be

shared among the processors thereby increasing the chances of cache miss. This problem can be dealt by using shared cache. Cache can be dynamically partitioned [3] and the thread that needs more workspace can be allocated more space, which would reduce the cache miss rate.

Shared cache also has larger hit access time and higher energy per access. Flexible cache sharing can also introduce complexities like requiring tremendous OS work when running multi-programs concurrently, increased hardware and architecture complexities like control complexity, coherence / consistency protocol complexity, and block replacement complexity. Private and shared cache configurations have the advantage of one as the disadvantage of the other and vice versa. Hence they prove to be best candidates to be evaluated and compared.

Further, cache architecture and memory technology can be selected to improve access latency. Alternately slow cache with high hit-rates can yield the same or better speed-up than fast cache with low hit-rates. Hit-rate is predominantly impacted by the application memory access pattern, the cache organization, and the cache size. This fact can be used to build multi-level cost efficient cache hierarchies. Cache architecture and memory technology can be selected to improve access latency, data transfer speed and bandwidth. Even different phases of the same application may benefit from different cache configurations in each phase. Recent technologies have enabled the tuning of cache parameters to the needs of an application. Core-based processor technologies allow a designer to designate a specific cache configuration. Additionally, processors with configurable caches are available that can have their caches configured during system reset or even during runtime. Such configurable caches have been shown to have very little size or performance overhead compared to non-configurable caches [4].

In this paper we compare the two cache configurations – private and shared by addressing the two microprocessor challenges mentioned earlier. The performance and power consumption characteristics for various cache configurations are simulated and analyzed.

3 Methodology

Architecture simulated:

The design of L2 cache has become very important because of the large die area it occupies. There is a tradeoff between cache latency and hit rate. Larger caches have better hit rates but longer latency and power dissipation. Cache hierarchy used here is separate L1 data and instruction cache. L2 unified cache private and shared.

Symmetric multiprocessor (SMP) is the most common architecture used today. It is a system with multiple identical processors that share main memory and controlled by a single OS instance. The multi-core used for analysis in this paper is SMP. It is created by duplicating the entire processor core with almost all of its subsystems on a single die and they often have caches sufficiently large to accommodate most of their working sets.

Simulators Used:

(1) Multi2Sim 2.3 which is a simulation framework modeling superscalar, multithreaded and multi-core processor [5]. It can run programs compiled for the x86 architecture. Memory hierarchy with the MOESI cache coherence protocol. Interconnection network simulation with bus and P2P topologies.

(ii) ACTI 6.5 which is integrated cache access time cycle time area, leakage, and dynamic power model [6].

Multi-core performance fully harnessed by parallel workload:

The full potential of multicore processors can be harnessed when the application running on them shows parallelism. Ideally *n* cores can yield *n* times the performance. However, this only applies to applications with inherent parallelism; multicore performance on a single sequential application might be worse than that of a high-powered sequential CPU.

To achieve a speedup of 80 with 100 processors according to Amdahl's Law the fraction of the original computation that can be sequential is 0.25%

$$80 = \frac{1}{\frac{\text{Fraction}_{\text{Parallel}}}{100} + (1 - \text{Fraction}_{\text{Parallel}})} \quad (1)$$

$$\text{Fraction}_{\text{Parallel}} = 0.9975 = 99.75\%$$

Today's applications and workloads have ample parallelism Many applications like networking (like IP forwarding), wireless (Viterbi decode and FIR filters), security firewalls (AES), and automotive applications (engine control) demonstrates parallelism properties. Emphasis on parallel programming is increasing as the performance improvement due to hardware advances are slowing down. Hence the performance and energy analysis is done with parallel work load on the processors.

Same amount of workload is allocated to each core. For example, to tune a cache for quad-core processor, four threads each comprising of same workload is created and one thread is run per core. This enables the energy consumed to be normalized per core. Also, the creation of independent process threads can reduce the interaction of threads on shared data and also will reduce the impact on hit rate due to inter processor communication. By eliminating the effect of inter

processor communication, the effect of data bandwidth between on-chip cache and off-chip memory with the increase in number of cores can be analyzed clearly.

Benchmark

PARSEC - Princeton Application Repository for Shared-Memory Computers has programs that focus on emerging workloads and represents next-generation shared-memory programs for chip-multiprocessors [7]. Bodytrack benchmark chosen has working set no larger than 16 MB and well suited for cache capacity of the latest generation of SMPs.

Procedure

Optimal cache size selection is based on the fact that too large of a cache results in cache fetches consuming excessively high energy and access latency [8]. Too small of a cache could also result in wasted energy due to thrashing in the cache. By increasing the associativity for certain applications the hit rate can be improved. In some applications, the higher associativity would cause wastage of energy as set-associative cache consumes higher power than direct-mapped cache. When dealing with many memory accesses with sequential addresses large caches line size are advantageous. Miss penalty, is increased since the data brought into the cache is more. As the line size increases, the number of blocks will decrease.

Applications require highly diverse cache configurations for optimal energy consumption in the memory hierarchy. Cache memories should satisfy certain performance bound requirement by the applications while the power is being lowered. In short size, associativity and line size of the cache should reflect the working set of the application.

Various cache configurations are simulated for Uni-core, dual-core, quad- core and octa-core with private and shared configurations respectively. Cache size, cache line size (32 bytes, 64 bytes) and associativity (direct mapped, 2 way associative, 4 way associative and 8 way associative) are varied and performance and energy tradeoff are studied for workload *bodytrack* benchmark.Performance bound energy efficient L2 cache configuration for the workload is selected by analyzing static energy consumption, dynamic energy consumption, total energy consumption and average memory access time. From these data Energy-Delay tradeoff characteristics is generated and analyzed. The most energy efficient cache configuration under best application performance is chosen. This is done by using the performance and energy metrics.

Performance Metrics

Average memory access time is used as the performance metric. This parameter is closely related to the actual number of cycles if the application is memory-intensive. The average memory access time is calculated according to the formula

$$\textit{Average memory access time} = \textit{hit time} + \textit{miss rate} \times \textit{miss penalty} \quad (2)$$

Hit time is calculated from cacti while miss rate is calculated from multi2sim. The miss penalty is 40 times longer than the hit time [9].

Energy Metrics

The total energy is the sum of dynamic and static energy given by the equation

$$E_{total} = E_{dynamic} + E_{static.} \quad (3)$$

Dynamic energy is dependent on the total number of cache accesses and the number of cache misses and the energy per-access. The energy consumption for a miss includes the energy for accessing off-chip memory, energy for the microprocessor when it is stalled due to cache misses and energy to fill the cache with a new block. The energy due to cache miss is 100 times the energy spent for a hit [9]. Static energy is calculated using total number of cycles and static energy consumed per cycle. Total cache access is calculated using multi2Sim while the static energy is calculated from cacti. As the cache size increase the static energy per cycle ($E_{static\ per\ cycle}$) increases and thus there is an increase in total static energy consumed by the benchmark.

Different energy-performance graphs are generated for different cores, cache sizes, block sizes and associativities. From these graphs i.e. total energy and average memory access time (AMAT) graphs, optimal cache configuration is chosen for each type of core. The process of selecting the optimal configuration involves the selecting the range of cache size for the block size and associativity where the AAT is low. Then from the range of cache size the optimal cache size having the least energy value is selected.

Energy delay metrics is given by the product of energy and average memory access time. These graphs are useful in choosing the optimal cache configuration for which the power consumption and delay are minimal. The cache configuration having the minimum energy delay metrics is the optimum cache configuration.

4 Results and Analysis

For all the cores, block size 64 showed good performance characteristics and their corresponding energy consumption values scored better than the 32 byte line caches. Associativity 2 and 4 showed good energy delay metrics when compared to

the other two associativities. The cache size for which the performance and energy are optimum are selected and tabulated in Table1 and Table 2.

Table1 Cache configuration optimized for performance and energy - Associativity 2

	Associativity 2		
	Size (kB)	**AAT ns**	**Energy (nJ)**
Unicore	16	1.5231	7671.3896
Dualcore Private	16	1.2094	7649.8689
DualCore Shared	32	2.2008	39427.325
QuadCore Private	32	1.669	30747.334
QuadCore Shared	64	2.8625	80345.983
Octacore Private	64	1.663	62220.956
Octacore Shared	128	4.0208	286938.77

Table2 Cache configuration optimized for performance and energy - Associativity 4

	Associativity 4		
	Size (kB)	**AAT (ns)**	**Energy (nJ)**
Unicore	32	1.9257	8043.3855
Dualcore Private	32	1.8775	11668.27
DualCore Shared	64	2.393	50849.715
QuadCore Private	64	1.8987	23778.883
QuadCore Shared	128	2.9187	87552.418
Octacore Private	128	1.8625	46865.048
Octacore Shared	256	3.858	390748.06

Cache Size

The scaling of cache size in private L2 cache is linear but the shared caches show a super linear increase. This signifies that the required cache area increases super linearly with increase in number of cores.

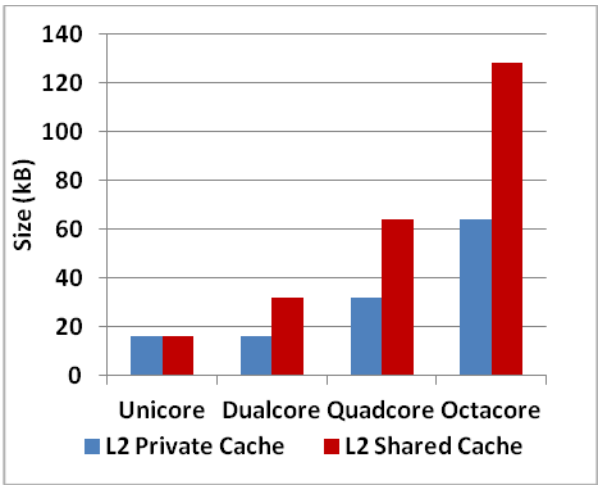


Figure1: Cache size trend for private and shared L2 cache increasing number of cores

Average Memory Access Time (AAT)

In the graphs above private cache seems to perform better. The AAT seems to increase linearly for shared cache while private cache seems to have a very slight increase in AAT with increase in number of cores.

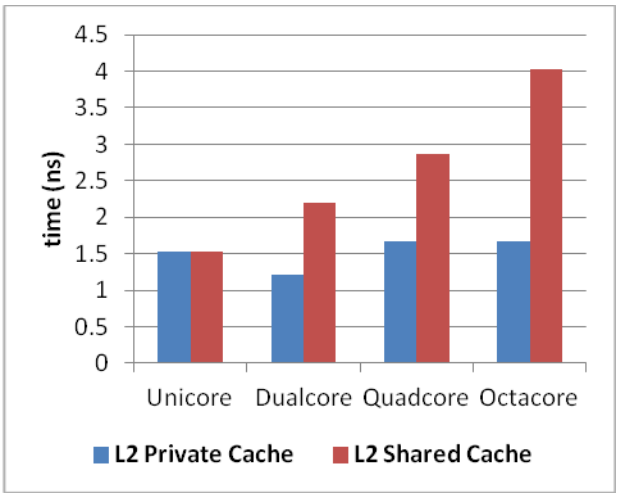


Figure2: AAT trend for private and shared L2 cache with increasing number of cores

For shared cache the access latency increases as its size keeps doubling with number of cores. Total L2 cache size for the private cache configuration doubles with increase in the number of cores but the private L2 cache dedicated to each processor is of the same size (8k). L2 cache access latency for the private partition remains the same as the number of cores

increase. As there is less true sharing of data between the different threads most of the L2 cache accesses belong to the private cache partition Another reason for this trend is the low memory bandwidth for shared cache and as the private cache can supply more data to the processor cache misses are reduced. Miss penalty is approximately 40 times that of hit time.

Energy Consumed

For shared cache energy increases exponentially with doubling the number of cores and doubling the workload. For private cache the energy increases linearly. The reason for this is the high energy consumed for a miss and hit for a larger cache when compared to a smaller cache. As the total area of private cache is divided into as many blocks as there are cores, the effective capacitance for the private cache block is less than the total capacitance of a shared cache. Hence the energy consumed for miss and hit for a private cache is less. Another reason for more energy spent by a shared cache is due to the increased cache misses due to the bandwidth reason as mentioned earlier. Energy consumed for a miss is approximately 100 times when compared to the energy consumed for a cache hit.

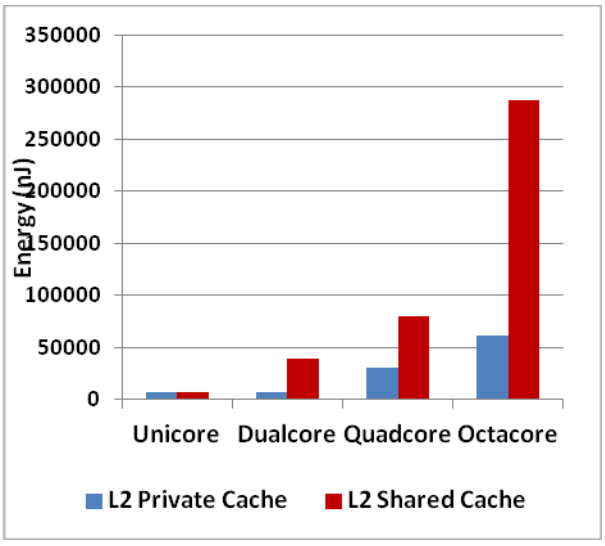


Figure3: Energy consumption for private and shared L2 cache with increasing number of cores

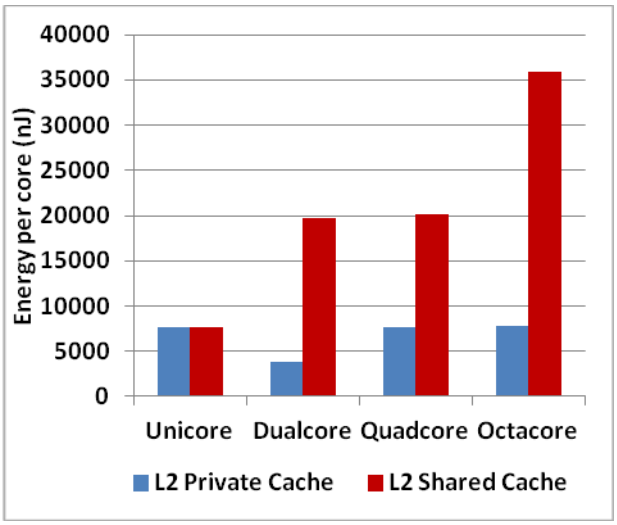


Figure 4: Energy consumed normalized per core

Again the private cache scores better in terms of Energy per core as the energy hit is smaller for smaller cache sizes. Banked cache can be used to reduce energy consumption in shared cache to reduce this effect.

Private L2 cache size is half the size of shared L2 cache for all the four types of multi-core processors i.e. 50 % savings in cache area. In the case of parallel threads with no data sharing there are more conflict misses in a shared cache than a private cache. For set associative or direct mapped block placement strategies, conflict misses occur when several blocks are mapped to the same set or block frame, also called collision misses or interference misses. This is dependent on the degree of data shared between the threads i.e. one thread can replace the data needed by another thread. Using higher associativity can minimize this effect but higher the associativity, higher the access time and counteracts the improvement provided by it.

Performance bound energy efficient cache size for private cache configuration for dual-core, quad-core and octa-core are 16k, 32k and 64k respectively which translates to dedicated 8k L2 cache block per core. So the access latencies for these blocks are the same. Work load on each processor is the same and the cache miss rate should be approximately same. From the graph it can be seen that a quad-core has a little more delay than the dual core. As the processes on each processor is independent the coherence complexities is mainly due to false sharing rather than true sharing.

The energy consumed by private L2 cache for dual core is only 20% of the energy consumed by shared L2 cache. For quad-core private L2 cache consumes 38% of that consumed by shared L2 cache. Octa-core private cache consumes 22% of shared L2 cache. Average of 27% of shared cache's total energy being consumed by a private cache. Energy saved is 73% in the case of private cache when compared to the shared cache. This ideal environment power savings percentage

shows a good scope of energy savings in the real time environment.

5 Conclusion

In this paper various cache organizations are simulated for multi-core processors and their performance and energy tradeoff are studied for emerging workload. The trend in performance and energy consumption for the optimized cache configurations with increasing number of cores is also analyzed.

For parallel workloads with little data sharing when the number of core is small, shared cache can be used and performance and energy dissipation as close as or better than private cache can be achieved by using techniques such as dynamic cache tuning and dynamic cache partitioning. As the number of cores increase the energy consumption increases exponentially and the performance increase linearly for a shared cache system. This is mainly due to the increase in mismatch between the processor data consumption and memory data supply rate. For a single memory multiprocessor this difference can be eliminated by using private cache and good off-chip memory to on-chip network. To make this type of multiprocessor design work efficiently, cores should be allocated the most independent processes that tend to share less data. Thus as the number of cores increase in a chip multiprocessor private L2 caches should be used instead of a shared cache when the workload shows ample parallelism and the threads don't share much data.

6 References

[1] Jurgen Reinold, "Performance Implications of Next-Generation PowerPC Micorprocessor Cache Architecture", IEEE Proceedings of COMPCON '97.

[2] A. Agarwaland M. Levy, "Going Mulitcore Presents Challenges and Opportunities", EE Times Design, http://www.eetimes.com/design/ signal-processing-dsp/4007064/Going-multicore-presents-challenges-and-opportunities.

[3] Ed Suh, L Rudolph, S Devadas, "Dynamic Cache Partitioning for Simultaneous Multithreading Systems", Proceedings of International Conference on Parallel and Distributed Computing and Systems 2001, August.

[4] Michael Zhang and Krste Asanovi´c, "Victim Migration: Dynamically Adapting Between Private and Shared CMP Caches".

Implications", Princeton University Technical Report TR-811-08, January 2008.

[5] R. Ubal, J. Sahuquillo, S. Petit and P. L´opez, "Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors", Proc. of the 19th Int'l Symposium on Computer Architecture and High Performance Computing.

[6] N Muralimanohar, R Balasubramonian, N P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches", Published in International Symposium on Microarchitecture, Chicago, Dec 2007.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh and Kai Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications", Princeton University Technical Report TR-811-08, January 2008.

[8] Paul Sweazey and Alan Jay Smith, "A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus", 1986, IEE Transactions.

[9] Y Leipo, S K Lam, T.Srikanthan, Wu Jigang "Energy Efficient Cache Tuning with Performance Bound", Proceedings of the Third IEEE International Workshop on Electronic Design, Test and Applications (DELTA'06).

[10] A Gordon-Ross, F. Vahid and N.D.Dutt, "Fast Configurable-Cache Tuning With a Unified Second-Level Cache", IEEE Transactions on Very Large Scale Integration (VLSI) systems, vol. 17, no. 1, January 2009.

[11] P. Stenstrom, "The Paradigm Shift to Multi-Cores: Opportunities And Challenges", Appl. Comput. Math. 6 (2007), no.2, pp.253-257.

[12] William James Dally and Brian Towles "Principles and Practices of Interconnection

Networks", Morgan Kaufmann, 2004.

[13] Ian K. T. Tan, I Chai, P K Hoong, "Pthreads Performance Characteristics on Shared Cache CMP, Private Cache CMP and SMP", 2010 Second International Conference on Computer Engineering and Applications.

[14] Ed Suh, L Rudolph, S Devadas, "Dynamic Cache Partitioning for Simultaneous Multithreading Systems", Proceedings of International Conference on Parallel and Distributed Computing and Systems 2001, August.

Simplified FPGA Design with Robei

Guosheng Wu
Robei LLC, Henderson, NV, USA
robei@robei.com

Abstract – *Robei is a tiny cross platform FPGA design tool that aims to simplify design procedure, transparent intellectual properties and reduce complexity. It makes FPGA design like playing with boxes by breaking down hardware into three basic elements: module, port and wire. Through these elements, engineer can implement either top-down or bottom-up design. Standard Verilog code can be integrated with other EDA tools generated from design diagram. Robei also runs on embedded platforms, which makes it distinctive from other EDA design software.*

Keywords: Robei; FPGA; Verilog; EDA; Simulation;

1 Introduction

FPGA is a technology intensive field full of innovations. Because of its low cost, easy to change and short time to market advantages, plenty of companies and individual engineers choose it to prototype products. Hardware engineers are focusing on developing their new designs in programming languages like VHDL and Verilog [1].

Although the future is very bright, there are still some obstacles that prevent FPGA market from fast growing, like high background requirement, opaque intellectual properties, and huge, complexity design tools. First of all, FPGA design requires not only knowledge of physics and circuits, but also digital design and logic synthesis. However, most of such knowledge is offered in universities and graduate schools, which limits the market growing. Second, FPGA tools are unique to each other. In order to implement a project on certain FPGA chips, the designer must stick to FPGA vendor's software and intellectual properties. Design engineers need to spend a lot of time to get familiar with these tools for the first time. Third, FPGA tools are huge in size and complex in contents. Current size of FPGA design software already counts in gigabytes. Bugs are increasing as the software size increase, which lead to the phenomenon that most engineers choose to use older version of design tools. Because there are so many options and features in design tools, which already beyond engineers' play-to-learn ability, so FPGA vendors need to spend a lot of time and money on training customers.

On the other hand, the successful stories of Apple's iOS and Google's Android [6] on mobile platforms proved that user interface become more and more important for customers. A lot of software companies already transferred their products from personal computer to mobile platform. However, FPGA design on mobile platform is still a challenge due to the high complexity and huge size of FPGA design tools.

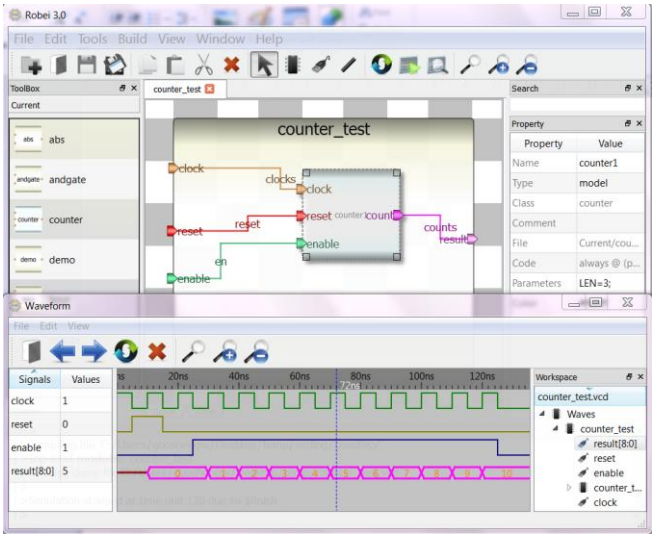


Fig.1 Robei user interface

This paper proposes a cross platform FPGA design software named Robei, shows in Fig.1. It is based on the most popular cross platform GUI framework, QT [7] which is already ported to iOS and Android platform. Robei can be re-compiled on many platforms without much modification. It aims to simplify user interface for FPGA design, transparent intellectual properties and reduce design complexity. The modern user interface of it combines diagram design method to represent circuit connections and coding method for algorithm inputs. Robei is designed to be as simple as possible for engineers. Let them mange in 15 minutes as long as they are familiar with Verilog language. Any pre-designed and system provided models are transparent to users. Property editor offers the most convenient method for viewing and modifying properties for each element. There are only three elements to representing the circuits: module, port and wire. By simply playing with these elements, FPGA designers can construct project by either bottom-up or top-down mechanism easily.

The remainder of this paper is organized as follows: element description is illustrated in Section 2; Code

generation is presented in Section 3 and a simple FIR filter design example is depicted in Section 4; screen captures of Robei running on Android platform is shown in section 5; conclusions and future work are given in Section 6.

2 Robei Elements

Robei employs three elements to represent Verilog components in hardware design: module, port and wire. In Verilog, circuits are represented by a set of "modules". A module may be only a gate, a flip-flop, a register, but also can be an ALU, a controller or a SOC system. We can consider a module as an abstract chip, which have different ports (pins) to communicate with other chips. A finished design module can be considered as model, which locates in "Toolbox" area and can be reused. Port is the interface channel for each module and model. Wire is used to connect ports on different module or model for signal transmission.

2.1 Module

In Robei, a module, the basic in a design, can be considered as a black box. Inside this box, designer can place ports algorithm codes and models. Each module can have zero or more ports for communication with other modules or models. The code view tab allows engineers to add or modify algorithm code easily to realize certain behaviors.

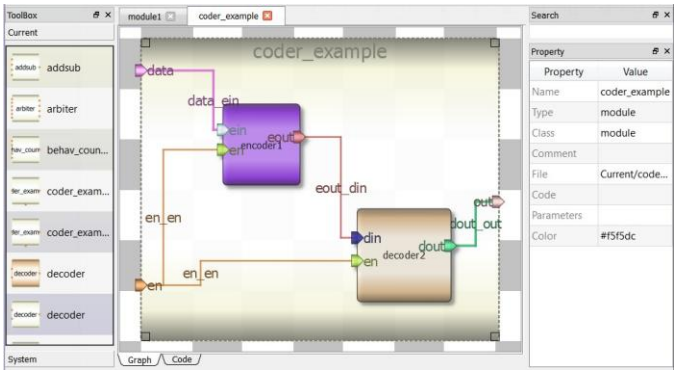


Fig. 2 module and model: only the biggest rectangle (coder_example) is module, the other smaller rectangles and rectangles in "ToolBox" are all models, which are pre-designed modules.

Based on usage status, a module can have different types. Currently under developing one has the type of "module", but once it is used in other modules, the type will automatically change to "model" as only certain properties can be modified in order to keep consistency with previous design. Fig.2 illustrates "module" and "model". There is another special type named "testbench", which is the top level design with stimulate code for simulation.

2.2 Port

A port may correspond to a pin on a chip, an edge connector on a board, or any logical channel of communication with a block of hardware. The detail properties are listed in fig.3. The type of port varies a lot as Robei supports many types in Verilog, like reg, wire, tri, supply, etc. There is "Datasize" option for port, which specifies the size of port if it is a bus. Some interesting features worth to mention are port can only slide on edges of module, and when module moves, port keeps sticking to edges all the time.

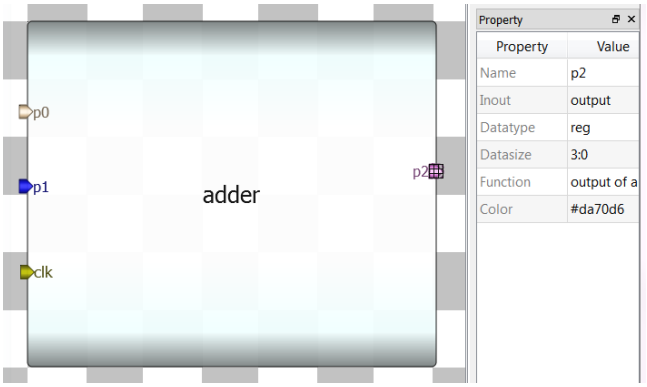


Fig. 3 Port properties

2.3 Wire

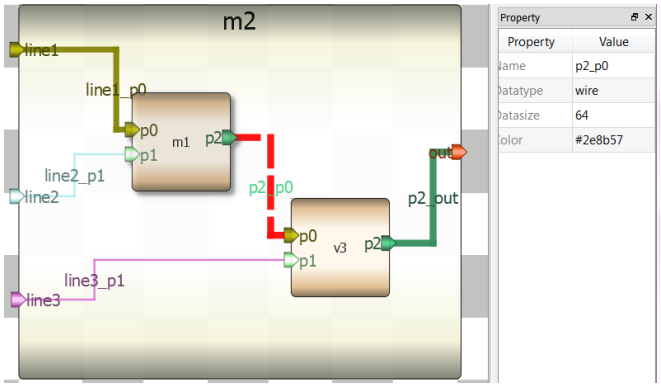


Fig. 4 Wire

Wire connects two ports and responses for signal transmission. Most of time, wire will inherent the color and data size from the first connected ports (as shown in fig.4). Based on different data size, wire has different thickness. Robei helps to check whether two connected ports have same data size or not when simulation start.

Module, port and wire are basic elements and used very often. Robei break down FPGA structure level design to these simple elements so even high school student can play with it.

3 Code Generation

The simplified user interface helps designer to reduce the code input and avoid mistakes. Instead of typing complete code for a project, hardware designers just need to write their core algorithm with code editor, while the interface design can be completed by playing with elements. For example, fig.5 and fig.6 show a simple counter design and its core algorithm that requires user to type in. The code under "Code start here" are core algorithm part. The other part like parameters, port declaration, module instantiation and extra signal declaration are generated based on designed property by Robei.

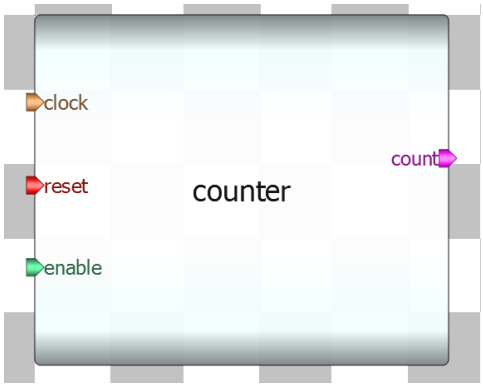


Fig. 5 Simple counter example

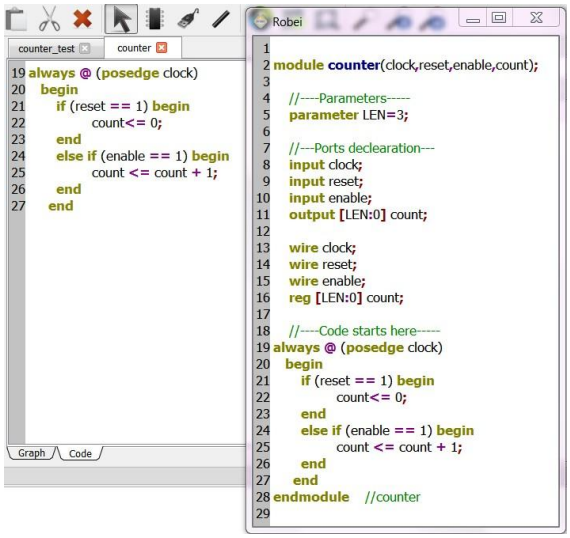


Fig.6 core algorithm that requires user to type in (on left), and generated code by Robei (right). Robei can generate code based the design diagram, while integrating with algorithm code from engineer.

Unlike Matlab toolbox [2], which encapsulates everything into its own model that makes custom model design complex, Robei displays every property (shows in property editor) to users. Users have the control of their own design. At the same time, model is transparent to users as they can view inside at any time by double clicking on it.

4 FIR Filter Example

Finite Impulse Response (FIR), which is weighted summations of input sequences, has been widely used in digital signal processing because of its advantages of linear phase, inherently stability, less precision errors and efficient implementation [3, 4]. Equation 1 is mathematical representation of FIR filter.

$$y[n]=\sum_{k=0}^{L}h[k]\times x[n-k] \tag{1}$$

Where $x[n]$ is input data at discrete time n, L is length of FIR filter, $h[k]$ is filter coefficient and $y[n]$ is the output at discrete time n.

In this example, choose $L=3$, the other coefficients as $h[0]=1/4, h[1]=1/2$ and $h[2]=1/4$ to avoid floating point calculation. So the output sequence would be:

$$\begin{aligned} y[n] &= \frac{1}{4}x[n]+\frac{1}{2}x[n-1]+\frac{1}{4}x[n-2] \\ &= \frac{1}{4}(x[n]+2x[n-1]+x[n-2]) \\ &= f(n)>>2 \end{aligned} \tag{2}$$

Where $f[n]=x[n]+2x[n-1]+x[n-2]$. In order to simulate the result accurately, instead of analyzing $y[n]$ response to $x[n]$, this paper analyzes the response of $f[n]$ to input data sequence. Design structure $x[n]\rightarrow f[n]$ shows in fig. 7.

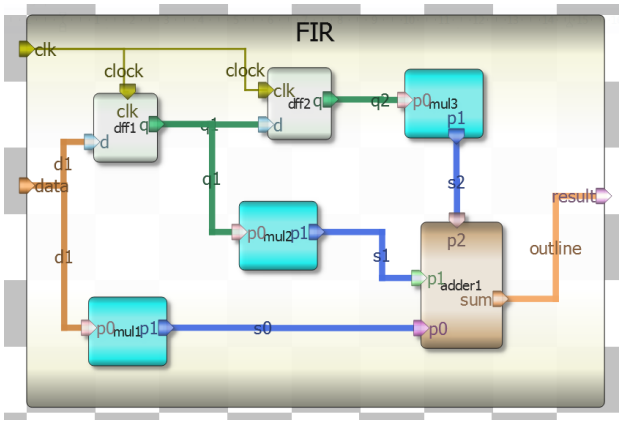


Fig. 7 FIR filter design example: there are two D flip flops to perform delay, and three multipliers with parameter of 1, 2, 1 correspondingly. The "adder1" performs summation of three inputs.

In order to verify the design, unit step response [5] is employed as stimulate for test bench. When $n=0$, $f[0]=h[0]$, and $f[1]=h[0]+h[1]$ if $n=1$. The

relation between $f[n]$ and coefficients shows in equation 3 when $n > L-1$.

$$f[n] = \sum_{k=0}^{L} h[k] \times u[n] = \sum_{k=0}^{L} h[k] \quad (3)$$

The output signal value proves statement above in simulation result (fig.8).

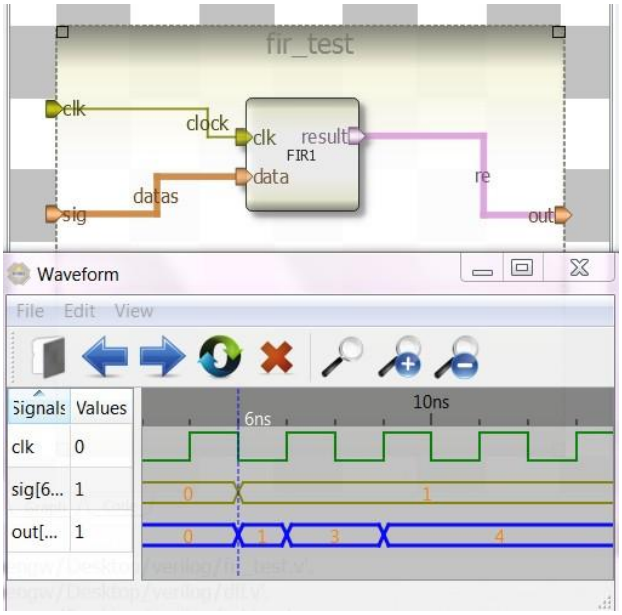
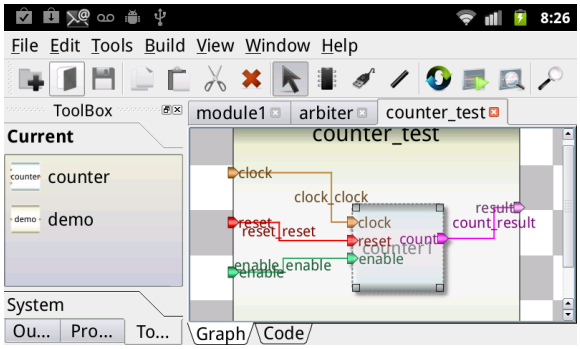


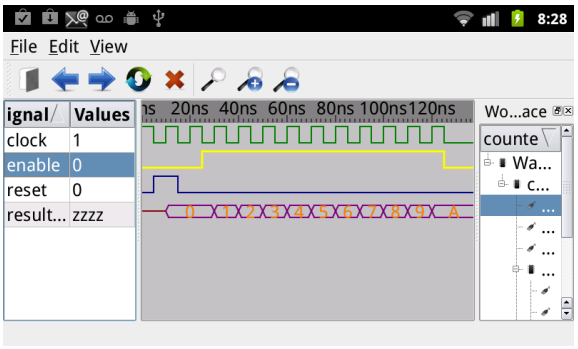
Fig.8 Step response of FIR filter

5 Embedded Platform

Thanks to Necessitas, which is the code name for porting of Qt to Android Operating System, and Verilog Behavioral Simulator (VBS) which is the first cross platform Verilog simulator designed by Jimen Ching, Robei is the first FPGA design tool that can perform simulation on embedded platforms like Android. Here are some images captured on android 2.3 platform when running. With Robei, developers can realize their design anywhere on mobile phones or tablets.



(a) User interface



(b) Waveform simulation

Fig. 9 Robei on Android

6 Conclusions

Robei starts a brand new way for FPGA design by providing simplified hardware design procedures, transparent property and incredible mobility. The goal of this design tool is letting everyone to play with FPGA design at anywhere and increasing the innovations from new sources other than hardware engineers.

The design concept of Robei is not only suitable for FPGA design, but also feasible in other fields. In future, It will be improved and extended for other research and industrial field to provide more practical values.

7 References

[1] Chinedu O.K., Genevera E.C., Akinyele O.O., "Hardware description language (HDL): An efficient approach to device independent designs for VLSI market segments," IEEE International Conf. on Adaptive Sci. & Tech. (ICAST). Abujia, Nigeria, pp. 262–267, Nov. 2011.

[2] Vila-Rosado D.N. and Dominguez-Lopez J.A., "A Matlab toolbox for robotic manipulators," Mexican International Conference on Computer Science. Guanajuato, Mexico, pp. 256–263, Sep. 2005.

[3] Chandra A., Chattopadhyay S., Sanyal S.K., "An efficient algorithm to minimize the number of coefficients of an FIR pulse-shaping filter," IEEE India Conf.. Kolkata, India, pp. 1–4, Dec. 2010.

[4] Jongsun Park, Woopyo Jeong, Mahmoodi-Meimand, H., Yongtao Wang, Choo, H., Roy, K., "Computation sharing programmable FIR filter for low-power and high-performance applications," IEEE Journal of Solid-state Circuits. Vol.39, pp. 348–357, Feb. 2004.

[5] Yutthagowith P., Pattanadech N., Kunakorn A., Phoomvuthisarn S., "Computer Aided-Program for Validation of Measuring System from Unit Step Response by Time Convolution Method," IEEE TENCON 2005, Region 10, pp. 1–6, Nov. 2005.

[6] Paul K., Kundu T.K., "Android on Mobile Devices: An Energy Perspective," IEEE Intern. Conf. on Computer & Inf. Tech. (CIT), Bradford, West Yorkshire, UK, pp. 2421–2426, Jul. 2010.

[7] Lobur M., Dykhta I., Golovatsky R., Wrobel J., "The usage of signals and slots mechanism for custom software development in case of incomplete information," Intern. Conf. On The Experience of Designing & Application of CAD Systems in Microelectronics (CADSM), Polyana-Svalyava, Ukraine, pp. 226–227, Feb. 2011.

SD-MARC: A New Multi-Processor Architecture

A. Somdip Dey
Department of Computer Science,
St. Xavier's College [Autonomous]
Kolkata, India.

Abstract *- In modern day, HPC (High Performance Computing) is applied to do massive scientific or huge computational works, but HPC architecture mostly consist of computer clusters (series of computers connected to each other to solve a single problem / task). Thus it is problematic for many institutions or organizations to maintain a computer cluster for doing huge computation. So, with the advancement of technology and efficient multiprocessor architecture, it is possible to integrate thousands of processors in one computer system and apply that system for doing the huge computation. This paper basically provides the idea of a new model to apply multiple processors (in order of n^2 (n x n) processors) in one computer system Architecture and how to implement the model to compute different problems faster than they can compute in reality; provided: n>=4 and n=2m, where m=0, 1, 2, 3, 4 ……. N.*

Keywords: Multiprocessor; High Performance Computing; Architecture; Matrix; Computer model;

1 Introduction

With the rising demand of Computational power, new architecture in multiprocessor system and technology advancement in multicore computers have been seen. This gave rise to High Performance Computing and parallel processor computer systems. Now a day huge processing power is required to compute and process a huge amount of data. So there is indeed an urgent need in advanced multiprocessor computer architecture to compute these huge amount of data within a short span of time.

The objective of this paper is to provide an overview concept of a new architecture in Multiprocessor System and how this architecture, SD-MARC, is beneficial in terms of computation and how to implement it.

2 The Architecture

This section covers the details regarding the architecture of SD-MARC.

Before moving on to SD-MARC, first let us see the various techniques and architectures used in multiprocessing systems.

Multiprocessor architecture can be basically classified into:

1.Symmetric Multiprocessing (SMP)
2.Asymmetric Multiprocessing (AMP)
3.NUMA (Non Uniform Memory Access Multiprocessing)
4.Cluster Multiprocessing (CM) [Concept of Distributed Computing]

SMP (Symmetric Multiprocessing):

In a multiprocessing system, when all CPUs / processors are treated equally, then the system is called Symmetric multiprocessing system (SMP). In SMP two or more identical processors are connected to a single shared main memory (computer memory) and are connected by a single Operating System instance, i.e. monolithic kernel type of OS (Operating System) is used for this type of system to utilize the resources. Now a day most of the multiprocessor architecture uses SMP. In SMP the processors are either connected to each other buses or crossbar switches or on-chip mesh networks.

The advantages of SMP include a large global memory and better performance per power consumption by the system. SMP also provides simple node-to-node (processor to processor) communication. The main disadvantages of SMP include the fact that the memory latency and bandwidth of a given node can be affected by other nodes, and cache "thrashing" may occur in some applications.

AMP (Asymmetric Multiprocessing):

AMP (Asymmetric Multiprocessing) designs uses SMP hardware architecture where a common global memory is shared between the various processors. In AMP designs, application tasks are sent to the system's separate processors. These processors may all be located on different boards or collocated on the same board, but each is essentially a separate computing system with its own OS and memory partition within the common global memory. One advantage of an AMP design is that asymmetric memory partitions can be assigned

from one large global memory, making more efficient use of memory resources and potentially reducing system cost.

SMP architectures differ from AMP in that a single block of memory is shared by the multiple processors or by multiple cores on a single multi-core processor. A single OS (Operating System) image runs across all the cores enabling truly parallel processing.

NUMA (Non Uniform Memory Access Multiprocessing):

In NUMA (Non Uniform Memory Access) multiprocessing the memory access time depends on the memory location relative to a processor. In recent time processors work faster than the memories used by them, so there is a big gap in the speed of a processor and a memory. So in a multiprocessor architecture to get high performance, one person have to install high-speed cache memory and use advanced algorithm to reduce the cache-'miss'. NUMA tries to solve this problem by providing separate memory for each processor, avoiding the performance hit when several processors attempt to address the same memory.

CM (Clustered Multiprocessing):

In Clustered Multiprocessing, many computers are loosely connected to each other, which forms computer cluster, and they work together so that in many respects they can be viewed as a single system. Usually the computers in a cluster are connected via high speed local area networks and this concept evolved from the concept of Distributed Computing, where different computers are connected to do one common task or achieve one same goal. In CM architecture we can see the use of master node and computing node, where the master node controls and distribute work (processes) to the compute node.

In SD-MARC, we combine the logic and ideas of most of these multiprocessor architecture, so that a new powerful architecture can be made out of it and can be implemented to compute faster and save cost of implementation relatively.

2.1 SD-MARC:

A. The Basic Design of SD-MARC

The basic design of SD-MARC consist of n^2 ('n x n') number of processors. Here, in this architecture the arrangement of the processors can be thought of square matrix formation of 'n x n', where 'n x n' signifies the square matrix system in which the number of columns and the number of rows are equal to 'n' and, n>= 4 and n is even number, i.e. n=2m , where m= 1,2,3,4,5,……..

So, from the above statement we get to know that the system consist of 'n x n' number of processors, as shown in the Fig 1.1, where the processors or the CPU (Central Processing Unit) are arranged in '4 x 4' formation and the total number of processors in this figure is 16 processors / CPU:

N.B.: In Fig 1.1 the number of processors in each row and each column are 4 and so it forms 4 x 4 multiprocessor system. From the figure Fig 1.1 we can see that 16 processors are arranged in 4 x 4 matrix formation and just like this 'n x n' number of processors can be arranged in the 'n x n' square matrix formation, where each row and each column contains 'n' number of processors.

Now this 'n x n' processor formation can be further divided into four divisions: Division 1, 2, 3, 4, where each division will have ((n x n) / 4) number of processors. So for example, if a system consists of 16 (4 x 4) processors then each Division will have 4 processors in it, or, for example in a system of 36 (6 x 6) processors there will be 9 processors in each division. This concept of Division system can be clear from the Fig 1.2, where the Division System of the Processors in 4 x 4 processor system has been shown.

Now along with this Division System we can use the concept of Computer Clusters. In Computer Clusters there is a concept of Master Node and Compute Node, where the Master node distribute the workload to different Compute node. Just like this concept, in the Division System of multiprocessor architecture, the concept of 'Master Processors' are implemented. Each division of processors has a master processor, which distribute the workload of that division to the different other processors, namely called Compute Processors, in that division. So basically there are 4 master processors in this multiprocessor architecture system.

Each Division has the processors numbered in order, like, CPU 1, CPU 2,…… and out of these processors, one is a master processor of that division, which is denoted as M CPU, as shown in Fig 1.3. So in the first division, i.e. Division 1, there will be CPU 1 to CPU j, where 'j' is the last processor number in that division and the starting number of processor in the next division will be CPU j+1, which goes up to CPU k, where 'k' is the last processor number in the Division 2, and so on.

In this system the 4 master processors are placed in side by side fashion, so that they can communicate with each other easily and very fast. The 4 master processors of 4 divisions communicate with each other time to time and synchronize among themselves the works / processes they are coordinating. The above mentioned concept can also be figured out from the Fig 1.3.

B. Processor Architecture and Relation between each Processor

Each processor will have high speed register memories of their own along with L1 (Level 1) Cache memory. Two adjacent processors / CPUs will be having or sharing a L2 (Level 2) Cache memory and then four adjacent processors will share L3 (Level 3) Cache memory. In Fig 2.1 we can visualize the concept clearly, where the relation between processors/CPU in terms of memory has been shown.

In modern world, the main computer memory can be classified basically in three ways:

a.Distributed Memory
b.Shared Memory

c. *Distributed Shared Memory*

Mostly Distributed Memory and Distributed Shared Memory concepts are used in case of computer architecture, where each CPU has a private memory block of its own.

But in this architecture a concept of '**Hybrid Memory Distribution**' is used. The 'Hybrid Memory Distribution' concept is: A memory block will not be private only to one CPU but each memory block will be shared by four CPUs / Processors, as seen in the Fig 2.1.

Since, four CPUs will be sharing only one memory, so there can be possibility of resource holding, which may give rise to Dead-lock situation. So to do away with that, each memory block will have memory address of its own and that memory address will not be same in any way in any of the other memory blocks. For example, in Memory 1the starting address is 1000 (Hex Address) and the end address in the memory is F000 (Hex Address), then the starting address in Memory 2 can be F001 (Hex Address) and the address in memory2 will never be the same as the addresses in Memory 1.

So from Fig 2.1 we can see that one memory is shared by four CPUs and these four CPUs form a 'Block'. In Fig 2.1 we can see two blocks of CPUs, Block 1 and Block 2. Basically in this architecture the whole multiprocessor system is divided in 'matrix' system, then each matrix system is divided in four 'Divisions' and at last each division has several Blocks.

Now each block has a memory of its own which is not shared with other blocks or other CPUs of other blocks, and the addresses in the memory never coincide with the address of other memories of other blocks.

If a matrix system of multiprocessor is of such a form that ((n x n) / 4) = ans, where ans is not perfectly divisible by 4 again then that system must have a sinlge CPU / processor left out of the block formation. For example there are 36 processors in a system then it is of form '6 x 6' CPU system. Now if we form four Divisions then there will be 9 CPUs in each Division. And if we try to form Blocks out of each Division then we can see that only two blocks can be made in each Division and there will be one CPU left out of the block in that Division. Then in that case we will consider the last CPU as the Master CPU and will attach that CPU to the last block formed in that Division. This concept can be clear from the Fig 2.2.

N.B.: In Fig 2.2 The 'BUS' means the Bus system through which each of the interaction of CPUs and memory system takes place.

Since '6 x 6' multiprocessor system is taken as an example so as from Fig 2.2 we can see that these two Blocks are in each Division and so there are 8 Computer Memory in the architecture.

C. Interaction Between the Processors

In Fig 3.1 the interaction and communication between the processors are shown. In this multiprocessor architecture, all the Compute processors in a Division are connected to the Master processor. The Master processor of that Division is again connected to other Master processors adjacent to it. The Master processor of a Division controls and distribute workloads to other Compute processors, and the Master processors then communicate with each other to synchronize the tasks they have performed. When a Master Processor is assigned a work (process) to perform, it first breaks that work into several small processes (can also be denoted as 'threads') and assign these small processes to different Compute processors of that Division along with the instructions of fetching different memory addresses needed to perform the small processes. The Master processor keeps track of each task (thread) assigned to each Compute processor and the memory addresses accessed by those Compute processors. After the completion of each task (thread), each Compute processor synchronize their tasks (threads) with the master processor and free the memory addresses used for the threads, which are again being tracked by the Master processor. And then that Master processor synchronize the process completed by it with the other Master processors of other Divisions.

D. Operating System to this Architecture

To utilize the computational capacities of the hardware of a computer an Operating System is most important. A kernel is a central component of an operating system. It acts as an interface between the user applications and the hardware. In most of the Computers in the world, either Micro Kernel or Monolithic Kernel type of Operating Systems is used. Monolithic Kernel can perform all the operations and consist of only one layer, i.e. in PCs (personal Computers) monolithic kernel is used to perform computation. And Micro Kernel can perform mainly few operations along with one global operation, i.e. it can perform one global process and other small low-level processes. Micro kernels are mainly used in Distributed Systems or Multiprocessor Systems.

Since, in this Architecture multiple processors are integrated in one single computer, so use of either of the two Operating Systems only will not be feasible to perform the processes. To solve this problem, a new concept of Operating System is introduced along with this architecture.

The new concept is that the system will have a basic monolithic kernel. This monolithic kernel will again have many small kernels (just like the micro kernels) inbuilt within it. These micro kernels will perform small tasks along with a global task and will be assigned to each Master processor. Since micro kernel can perform only one global process at a time, the Master processor can utilize this fact and perform that one global process at a time along with small other processes. This type of a Kernel is called a 'Hybrid Kernel'.

3 List of Figures:

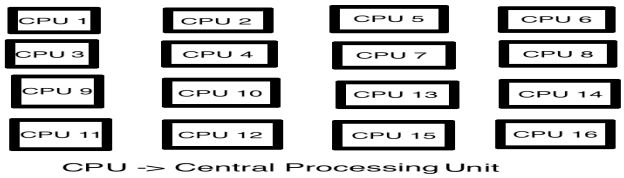


Fig 1.1: Figure shows 16 processors arranged in 4 x 4 matrix pattern.

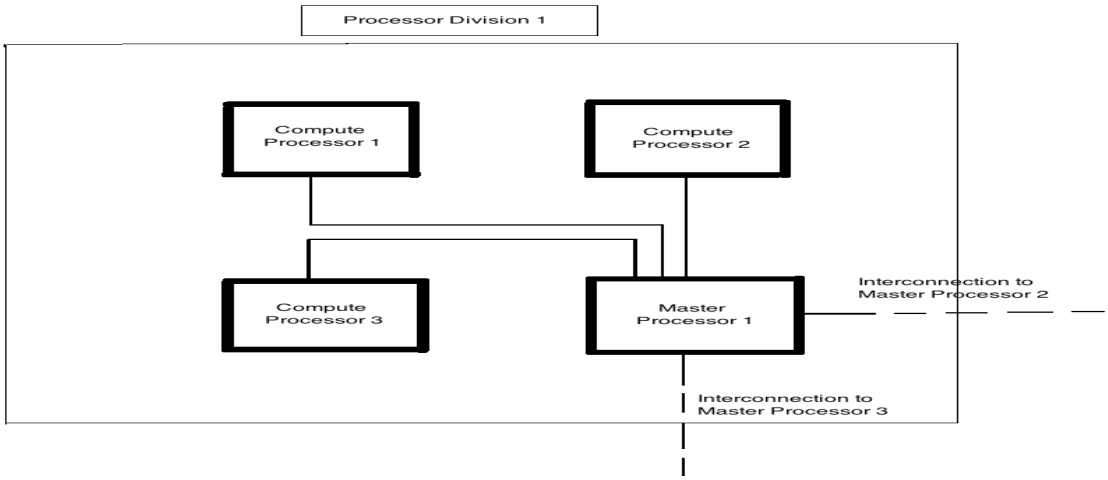


Fig 3.1: The Interaction between CPUs / Processors in '4 x 4' Processor System

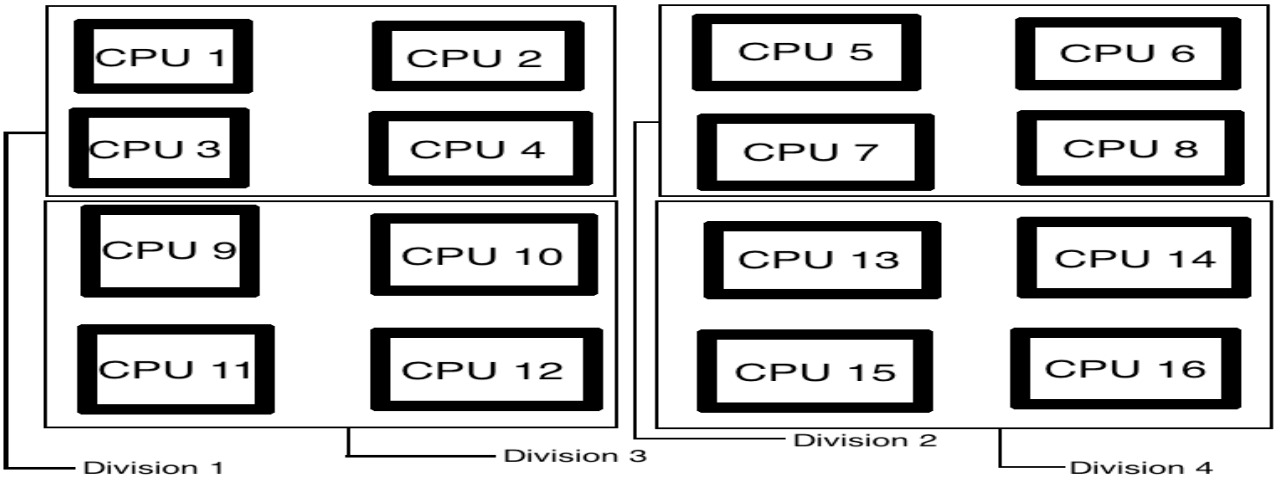


Fig 1.2: Division System in 4 x 4 Processor System

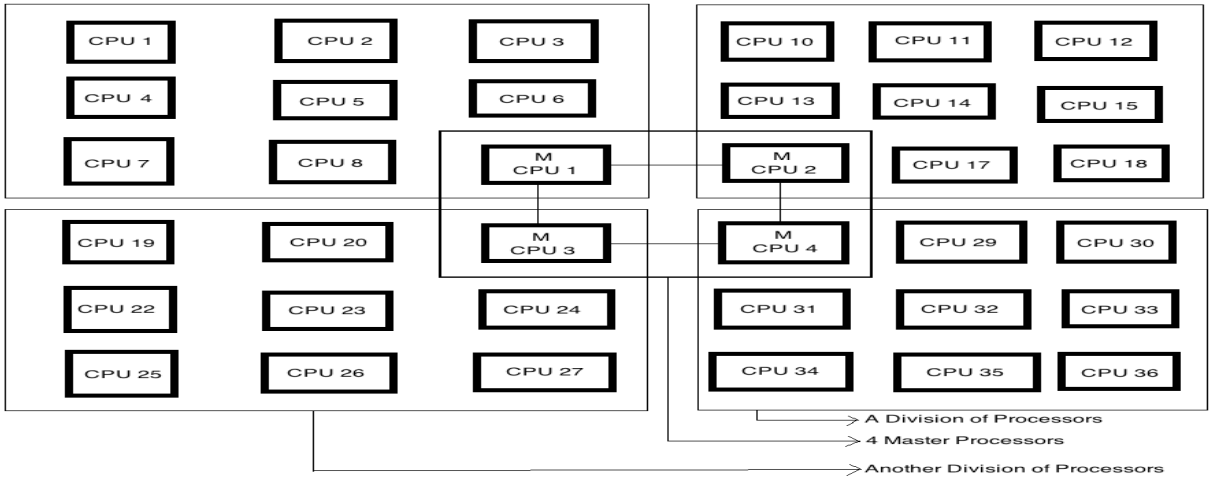


Fig 1.3: Division System with the concept of Master Processors and Compute Processors in (6 x 6) processor System

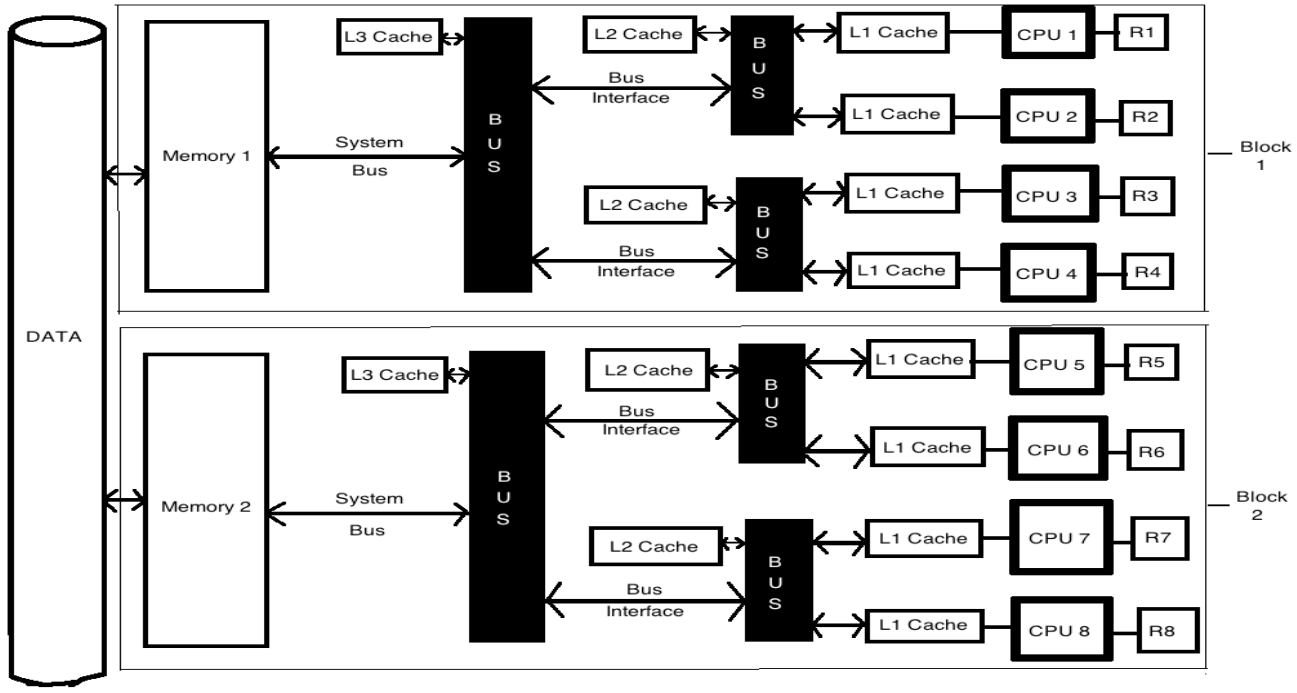


Fig 2.1: Relation between CPUs in terms of memory
N.B.: R1 – R8 are Register Memories of the CPU 1 – CPU 8

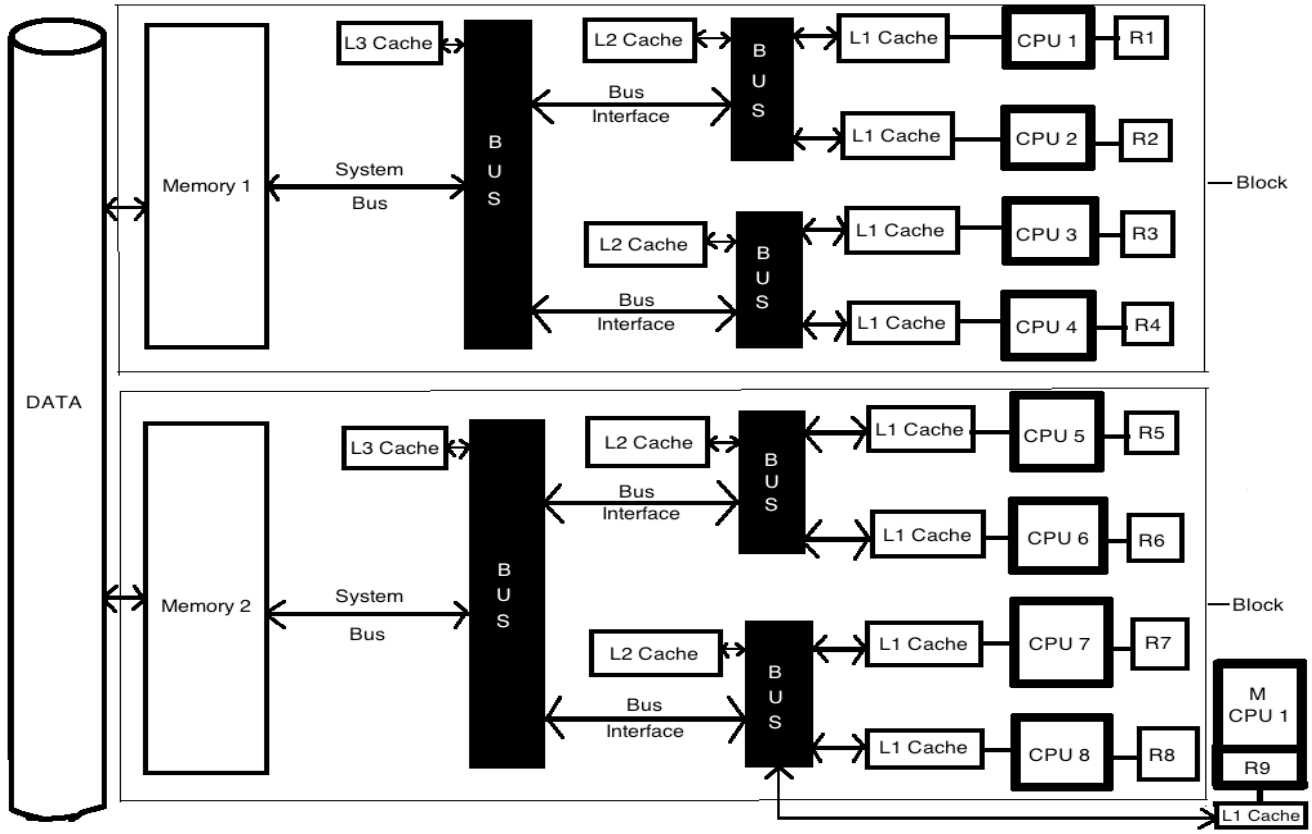


Fig. 2.2: Attachment of the M CPU (Master CPU) to the last Block in a division

N.B.: R1 – R8 are Register Memories of CPU 1 – CPU 8; R9 is the Register Memory of M CPU

4 Few More Details

A. *Cooling of the System*

Since multiple processors are integrated within a single computer, it is obvious that the system will get very hot and will generate a lot of heat. So to deal with this problem high level of cooling system is needed to be installed within the system. Both Air Cooling and Liquid Cooling systems can be used to keep the computer cool and ventilate out the hot air.

B. *Need of Multiple Processors in one System*

In HPC (High Performance Computing) or in Super Computers mainly the concept of Computer Clusters are used to compute a task, where many computers are connected to each other via a network. But there is an ever growing demand of fast computers in todays world. If Computer Clusters are used then the system become huge to maintain and are also very costly economically. So if multiple (in order of hundreds or thousands) processors are integrated in one computer using nano-technology or recent technological advancement then it will be easier to manufacture smaller (in size) super computers and will be easier to maintain. If multiple processors are used in one system then it will be easier to compute and finish a task faster than it could be, and to make the system easier to be maintained.

C. *Use of A Single Computer Memory for 4 CPUs*

In this Architecture four CPUs (processors) share a single computer memory (RAM or Random Access Memory in general). If the concept of Distributed Memory System or Distributed Shared Memory System is used then the number of processors will have equal number of computer memory, and in this architecture it is of the order 'n x n' (n^2) which is a very large number. So the number of computer memories used would have been n^2 for 'n x n' number of processors and it would have made the system pretty large to maintain. To deal with this problem four CPUs share a single computer memory and for a system with 'n x n' number of processors, the number of computer memories needed in this architecture is (abs[(n x n) / 4] * 4), where abs() denotes the absolute value or the integer value of a number, for example: abs(2.25)= 2.

D. Architecture Usage and Future Scope

Since, SD-MARC uses all the advantages of the different types of multiprocessor architecture system, it is beneficial to use and implement it, but still there's a drawback of this architecture. There's no Operating System till date to support this architecture. If this architecture is used in small systems or very few number of processors are used to built a multiprocessor system using this architecture, then there may be lack of performance, as it will be in a system with thousands of processors.

5 Conclusion

Since, this multiprocessor architecture, SD-MARC, is used in a single computer system, it can be used in any field of application, where intense computational power is needed in one single system. For example, this architecture can be used by hospitals to compute different problems and chemical structures of medicines, can solve and diagnose different diseases in very less amount of time. Even computer enthusiast can use this architecture to build their own Personal Super Computers and use those to compute time-consuming problems in really less amount of time.

6 References

[1] W. Anderson, F. J. Sparacio, and R. M. Tomasulo, ''The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling,'' *IBM Journal of Research and Development,* Vol. 11, No. 1, pp. 8-24, January 1967.

[2] Proceedings. Supercomputing '88 (IEEE Cat. No.88CH2617-9) ,November, 1988.

[3] Norman P. Jouppi, ''The Nonuniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance,'' *IEEE Transactions on Computers,* Vol. 38, No. 12, pp. 1645-1658, December 1989.

[4] Youngjin Kwon, Changdae Kim, Seungryoul Maeng, Jaehyuk Huh, "Virtualizing performance asymmetric multi-core systems", International Symposium on Computer Architecture, pp. 45-56.

[5] Rajkumar Buyya (editor): *High Performance Cluster Computing: Architectures and Systems*, Volume 1, ISBN 0-13-013784-7, and Volume 2, ISBN 0-13-013785-5, Prentice Hall, NJ, USA, 1999.

[6] Internet Source: http://www.intel.com/pressroom/archive/reference/whitepaper_QuickPath.pdf [last visited 10/03/2012]

[7] Internet Source: http://users.ece.utexas.edu/~bevans/papers/2009/multicore/MulticoreDSPsForIEEESPMFinal.pdf [last visited 11/03/2012]

A New Technique To Use A Parallel Compiler for Multi-core Microcontrollers

A. Somdip Dey
Department of Computer Science,
St. Xavier's College [Autonomous]
Kolkata, India.

Abstract - *Now a day multi-core microcontrollers are being used in various fields. Due to resource limitation of microcontrollers, programming them is difficult. This work presents a simple parallel compiler that can exploit multi-core to speed up parallel tasks on a multi-core microcontroller. The parallel constructors are introduced. A scheme to use compiler directives to hint the compiler is discussed. Experiments on the real processors are performed to validate the scheme. The results show that the compiler can exploit multi-core to speed up the computation tasks on the target microcontroller.*

Keywords: multi-core microcontroller; parallel compiler; processing; multi tasking;

1 Introduction

Microcontrollers are very interesting and fascinating due to their vast applications nowadays. They are used in many embedded systems from a tiny robot to a car and even an enormous airplane. New development of microcontrollers occurs at very fast pace. Currently multi-core microcontrollers become available. However, programming model for such platform is still rare, especially the parallel compiler. This lack of proper compiler has hindered the use of multi-core microcontrollers.

Many programming models exist for desktops and servers. For example, OpenMP [1], Data parallel [2], including Intel Threading Building Block [3]. They are not suitable to multi- core microcontrollers due to the limit of resource of the target architecture. In particular, we are interested in one cost- effective multi-core microcontroller from Parallax Inc., an eight-core processor called "Propeller"[4]. Its architecture is very unusual. It has a global shared memory with very slow access and fast but small local memory for each core (see Fig. 1). Its instruction set is tailored to some specific way of using individual core. All of these results in a rather difficult programming model for this architecture.

The chip architecture was designed to have the 8 parallel independent cores which named cog by Parallax Inc. There is 2KB internal memory or register in each core. The share memory of 32KB RAM and 32KB ROM are accessible through memory hub which grants access only a single core at a time starting from cog 0 to 7. All I/O pins are connected to every core. Input pins value can be read and output pins value can be written at any time but the output pins value is the logical "OR" value from all 8 cores as the output pins can be driven by those cores.

This work presents a compiler of a parallel language targeted for Propeller. The approach we use in this work is to take a simple imperative language (in this case, a simplified C) and add some decoration to present a parallel constructor for a block of sequential code. The compiler will recognize this decoration and generates proper parallel code for that section. Of course, this approach has the limitation that the type of parallel constructor is limited. We show a number of useful constructors and their applications.

The flow of the article is as follows. The next section discusses the target language, Spin [5], which is embedded with the processor. Section III explains the compiler. Section IV shows examples of the parallel constructors. The conclusion is given in the end.

2 Propeller Spin Language

Spin language is embedded as an interpreter in the microcontroller. This is the solution embedded with the chip. Its intention is to make available a language that is used to control and specify the parallel operations of its multi-core thus simplify how users can program many cores of the chip. The structure of a Spin program is as follows. A global variable is declared before functions then the constant declaration. Object modules, which are a kind of function library or the class in OOP concept, also can be declared for use as well. The main function has all local variables declared immediately after the function declaration. The body of the function then follows with the function's statement.

The Fig. 2 shows a simple program written in Spin to generate frequency specified by user at customizable starting output port 16. There is one global variable named "wait_delay" which will be initialized to be user delay and growing in each iteration. The loop is created by the command "repeat" with a local variable i which will be used to step the output port forward. The value of the pin port A which is the pin 0 to 31 is set by the register name OUTA. The port is specified by the array variable indexing. The counter register CNT is added to the delay time and put as a parameter of the "waitcnt" function to interrupt the hardware to stop.

Special commands are used to start and stop a Propeller core. The command "cognew" initializes a new available core by uploading the corresponding source code to be run (similar to spawning a thread). The other command "coginit" related to the core initialization command is used when programmer need to identify a specific core of its availability. Both commands require the parameter to be used as a stack memory which is used for temporary calls and expression evaluation when the core is starting. A suitable amount of stack space is necessary for a program on a core to properly run. Also to stop the running core, "cogstop" command is used. Fig. 3 shows an example for swapping the output using many cores. The coginit command is used to start the specified core with the swap function at the given pin port.

3 Compiler For Parallel Programs

The base-language for our compiler is a simplified C called RZ [6]. RZ is a small language. Its syntax is very similar to the language C (but without type). It is the language used in a teaching class about compiler. The full set of compiler source code and tools are available in our institution [7]. The new parallel operations are added into the language by using compiler directives scoped over a section of normal code.

"#pragma parallel for" is used to specify parallel operations over a for-loop body. The parameters in the for-loop head: initialization, conditional and increment of loop-index, are parsed and stored. They are used for generating the output parallel code. Here is an example of the use of the pragma (see Fig. 4).

The output of the compiler is the statements in Spin to distribute the work in the body of for-loop over the available cores. In Spin language, the command for iteration is "repeat" and to start a new process, the command "coginit" is used.

As the to-be-run parallel code is issued to many cores, the command to start each core is called. The number of calls is equal to the number of cores. The body of loop is generated as a parameterized function. The parameters of the loop are used as the parameters of the "coginit" command.

The code-generator generates the body of loop as a function with arguments for sharing task and the local variables. This function contains the loop with specified iteration (the output Spin code will be shown with examples in the next section).

3.1 Parallel Constructors:

To illustrate the idea of using #pragma in parallel programs, two constructors are discussed. Example of parallel programs and the output code from the compiler are shown. These examples are: matrix multiplication, reduction and odd-even sort. We believe these examples show the intended use of the parallel #pragma which can be applicable over a wide range of parallel programs.

3.2 Matrix Multiplication:

Fig. 5 shows the pseudo code for matrix multiplication (NxN). Initially C is zero. The number of calculation is growing rapidly when the size of matrix is increasing. When the matrix size grows from 2x2 to 3x3, the number of calculations is grown from 8 to 27. It is growing at the rate of N3 where N is the matrix size (NxN).

There are three nested loops. Each element of C[i][j] is the inner product of the row i of A and the column j of B. The parallelization is made at the deepest inner loop. By distributing the calculation to each core, it will reduce the execution time in proportion to the number of core used in the calculation. Here is how to write a parallel version of the matrix multiplication (see Fig. 6).

We distribute the different "C += A * B" over the different core. To put a large amount of work to the limited number of cores, two loops (the innermost) are required. The first loop is used to issue work to several cores and the second loop is used to strip the vector properly for each core. Here is the output of the compiler (see Fig. 7).

The innest loop is stripped over many cores (a constant CORE) and "coginit" is called for each core. The function "par_fun" is the body of the for-loop. The "@stack[32*l]" is required to allocate a stack space for the core.

3.3 Reduction Sum:

Fig. 8 shows the code of reduction sum. We use "#pragma parallel for reduction" to indicate the type of parallel constructor. Initially sum is zero. The vector V (of size N) is reduced to a scalar value by summation. Reduction is done by the divide and conquer method. The vector is divided into two halves. The operation is applied on a pair with one element from the first half and another element from the second half. The result is stored "in-place"

at the first half of the array. Each iteration reduces the vector by half if there is enough processors to perform the operation concurrently and there is no data dependency. It takes log2N to reduce a vector of size N to a scalar. Each pair of numbers can be processed in each core in Propeller. Using two cores, a vector of size 1024 can be reduced to a scalar value with 10 iterations. Here is the output Spin code for reduction sum (see Fig. 9).

The logarithm function is used to calculate the number of the iteration. Also the power function, it is used to find the size of each level of the tree to correctly distribute the work to the available cores.

3.4 Odd-Even Sort:

This is an algorithms used for sorting on parallel systems. The comparison operation can be done in parallel. The algorithm compares the adjacent elements. Assume the first round is an even-round. The comparison starts at the 0th element. The next odd-round starts at the 1st element. If there are N cores, in N-1 iterations the sorting is done. Fig. 10 shows the parallel code for odd-even sort. Assuming the index starts at 1.

The output Spin Code is shown in Fig. 11.

The first loop iterates over all items. The second loop iterates over half of the items because each iteration is dealing with odd-only or even-only. The iterations needed in each round is N/2. The third loop distributes the work over many cores starting from the odd or even index. The index is calculated using modulo function which can be seen as the double-slash symbol.

Each of the algorithms is compiled and the result contained the parallel code section. All codes have a dedicated function for "coginit" command to be called when starting and initializing a core to run in parallel mode.

4 List of Figures

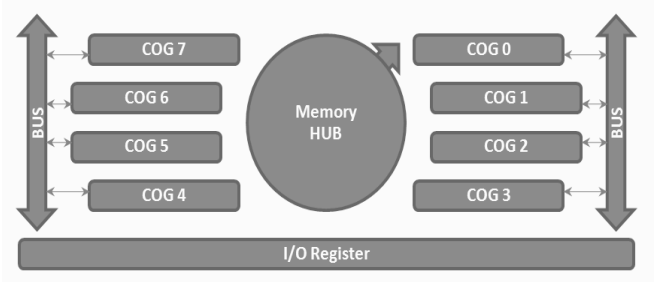


Fig. 1. The architecture of Propeller

```
CON
   OUTPIN = 16
   USER_DELAY = 1_000_000
VAR
   LONG wait_delay
PUB main | i
   wait_delay := USER_DELAY
   repeat i from 1 to 3
      OUTA [OUTPIN+i] := 1
      waitcnt (CNT+wait_delay)
      wait_delay += wait_delay
      OUTA := 0
```

Fig. 2. The sample LED toggling program in Spin language

```
PUB main
   coginit (1, swap (16), @_stack[0])
   coginit (2, swap (17), @_stack[8])
   coginit (3, swap (18), @_stack[16])

PUB swap (port)
   repeat
      OUTA[port] = !OUTA[port]
      waitcnt (CNT+1_000_000)
```

Fig.3. An example of a parallel program written in Spin

```
#pragma parallel for
   for (i = 0; i < 10; i = i + 1)
      a[i] = b[i] * c[i];
```

Fig. 4. The example of the pragma compiler directive

```
for i from 1 to N
   for j from 1 to N
      for k from 1 to N
         C[i][j] += A[i][k] * B[k][j]
```

Fig. 5. Pseudo code for matrix multiplication

```
for (i = 1; i <= N; i = i+1)
   for (j = 1; j <= N; j = j+1)
      #pragma parallel for
      for (k = 1; k <= N; k = k+1)
         C[i*N+j] += A[i*N+k]*B[k*N+j];
```

Fig. 6. The ready-to-compile parallel version of the matrix multiplication

```
repeat i from 1 to N
   repeat j from 1 to N
      repeat k from 1 to N/CORE
         repeat l from 1 to CORE
            stepsize := ((k-1)*CORE)+l
            coginit (l+1,
               par_fun (i, j, stepsize),
               @_stack[32*l])

PUB par_fun (i, j, k)
   C[((i-1)*N)+(j-1)] += A[((i-1)*N)
      +(k-1)]*B[((k-1)*N)+(j-1)]
```

Fig. 7. The matrix multiplication compilation output

```
#pragma parallel for reduction
   for (i = 1; i <= N; i = i+1)
      sum = sum + V[i];
```

Fig. 8. The parallel code of reduction sum

```
repeat i from LOG2(N) to 1
   repeat j from 1 to (POW(i-1)/CORE)+1
      repeat k from 1 to CORE
         coginit (k+1,
            par_fun (((j-1)*CORE)+k,
            POW(i-1)), @_stack[8*(k-1)])

PUB par_fun (idx, size)
   V[idx-1] += V[(idx-1)+size]
```

Fig. 9. The reduction sum output Spin code

```
for (i = 1; i <= N; i = i+1)
   #pragma parallel for
   for (j = (i mod 2)+1; j < N; j = j+2)
      if (A[j] > A[j+1])
         swap(A[j], A[j+1]);
```

Fig. 10. The parallel code for odd-even sort

```
repeat i from 1 to N
   repeat j from 1 to ((N/2)/CORE)+1
      repeat k
      from (i//2)+1+((j-1)*(2*CORE))
      to (j*(2*CORE)) step 2
         coginit (((k//(2*core))/2)+2,
            par_fun (k-1),
            @_stack[(k//(2*CORE))/2])

PUB par_fun (idx) | tmp
   if D[idx] > D[idx+1]
      tmp := D[idx]
      D[idx] := D[idx+1]
      D[idx+1] := tmp
```

Fig. 11. The odd-even sort output Spin code

5 Experiment

Three algorithms: matrix multiplication, reduction sum and odd-even sort, which can exploit the parallelism are used. Each program is compiled and then run in the Propeller microcontroller chip. The results are recorded for each specific test configurations related to number of core used.

The following figures listed, show the comparison of the execution time of each program varies by the number of core used. For the matrix multiplication, Fig. 12 shows that the speedup increases when more cores are used. Compare to single core, the speedup of the 6-core on 48x48 is 1.2. The larger matrix size have higher speedup as the overhead is smaller compare to the total time of execution.

The result of the reduction sum program is shown in Fig. 13. The result of the small data size indicates that the 6-core is slower than the 3-core. When increase data size, the 6-core speed up is better than the 3-core. The reason of anomaly, which the 6-core is slower on small data size is that there is larger overhead in core initialization process. The result of the odd-even sorting is shown in Fig. 14. For the 3-core, it seems that the speedup is almost independent from the size of data.

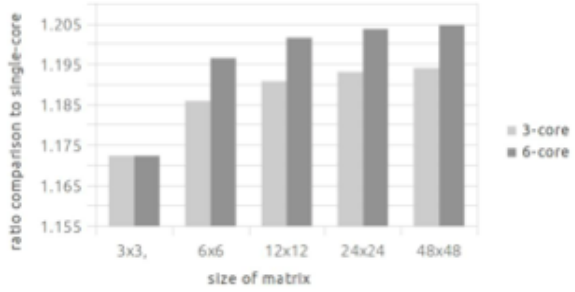


Fig. 12. Matrix multiplication execution time: multi-core vs. single-core

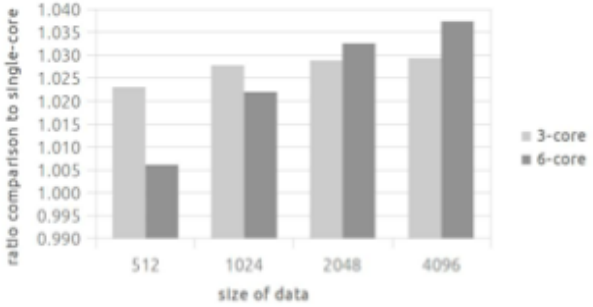


Fig. 13. Reduction sum execution time: multi-core vs. single-core

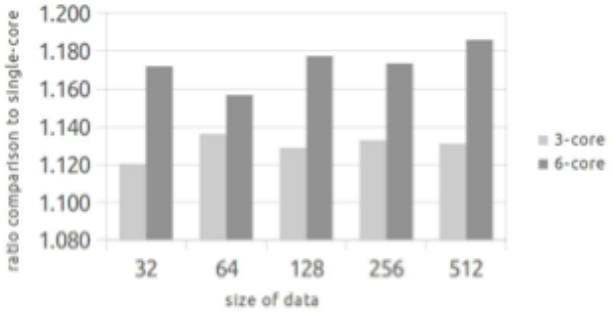


Fig. 14. Odd-Even sorting execution time: multi-core vs. single-core

6 Conclusion

This work presents a parallel compiler for a particular multi- core microcontroller. The compiler directive "#pragma" is used to hint the compiler to generate a proper code for parallel section. Two parallel constructors are introduced. The experiments are performed to measure the speed up of the execution time while varying the number of cores used. The results show that the compiler generates correct output code that can exploit multi-core to speedup the computation.

7 Acknowledgment

Somdip Dey expresses his gratitude to all his fellow students and faculty members of the Computer Science Department of St. Xavier's College [Autonomous], Kolkata, India, for their support and enthusiasm. He also thanks Dr. Asoke Nath, professor and founder of Computer Science Department of St. Xavier's College (Autonomous), Kolkata, for his constant support and helping out with the preparation of this paper.

8 References

[1] L. Dagum and R. Menon, "OpenMP: An Industry Standard Api for Shared Memory Programming," *IEEE Computational Science and Engineering,* vol. 5, no. 1, pp. 46-55, Jan-Mar 1998.

[2] W. Daniel Hillis, Guy L. Steele, Jr., "Data parallel algorithms," *Communications of the ACM - Special issue on parallelism,* IEEE Magazine, vol 29, no. 12, Dec 1986.

[3] N. Popovici and T. Willhalm. "Putting Intel Threading Building Blocks to work," *Proc. of the International Workshop on Multicore Software Engineering (IWMSE 2008),* Leipzig, Germany, May 11 2008.

[4] Parallax Propeller, http://www.parallax.com/tabid/407/Default.aspx, 2012. [ONLINE]

[5] Martin, J. and Lindsay, S., "Parallax Propeller Manual," 2006.

[6] RZ language and its compiler, http://www.cp.eng.chula.ac.th/faculty/pjw/project/rz3/index-rz3.htm, 2012. [ONLINE]

[7] RZ compiler tools kit, http://www.cp.eng.chula.ac.th/faculty/pjw/project/rz/rz2compiler.htm, 2012 [ONLINE]

Data Center Design and Implementation at the University

Askar Boranbayev[1], **Sergey Belov**[1]
[1]Nazarbayev University, Astana, Kazakhstan

Abstract - *Universities are trying to make their academic and business processes more efficient and effective. In our opinion the appropriate use of information technology may be an important source of future success for the university. This paper talks about one of the strategies for moving a university toward more effective IT using data center virtualization technologies, while building a modern data center. Virtualization is a technology to help data centers enable more agile operation process, increase availability, improve security, and possibly reduce cost.*

Keywords: Data center, virtualization, information technology, servers, network

1 Introduction

A university's fundamental technological base needs to be composed of appropriate modern IT hardware and software systems. We have designed and implemented a new data center at our university. It has a good physical security, with card-based access and tracking of all personnel entering the data center; video recording of all activity, good cooling system, raised floors, UPS power for the entire data center, latest blade server technology, using server virtualization, and information systems running on servers to automate various business processes and academic processes at the university.

This project was initiated to solve many of the IT infrastructural problems the university was facing such as backup and recovery services and other conditions. The design included the review and evaluation of experiences of other data center implementations and their infrastructure, and the development of a component selection and configuration. The design consists of a combination of proprietary and open source infrastructural solutions.

2 Virtualization

Usually a large percentage of university's expenditures involve IT. The university IT department had a task to implement high-performance, efficient and yet cost-effective solutions for the server hardware and software for the data center. We have looked at a variety of architectures of server platforms such as Itanium, SPARC, PA-RISC, x86, and we stopped on the x86 architecture because of the efficient, economical and fast-growing technology.

After studying the experience of other universities, we have understood that most high-performance servers are loaded only at 25-30 percent [1] when performing their daily tasks. This is why we have decided to emphasize on virtualization technology. Nowadays, the concept of "virtual machine" has ceased to be something unusual. Many universities have learned to use virtual machines in their daily operation to increase the efficiency of their IT-infrastructure.

The IT infrastructure virtualization is a process of representing a set of computing resources, storage resources, or any of the applications by combining them logically and by centralized management, which provides various advantages over the original standard configuration. [3][4] The virtual way to resources on-demand is not limited to the geographic location or physical configuration of the components. Usually virtualized resources include computing power, data storage, and efficiency.

Several useful aspects of this technology have been taken into account during the implementation:
- The use of multiple virtual servers on one physical can increase the percentage of server utilization up to 80 percent, while providing substantial savings on the purchase of hardware.
- Virtual servers are a very good solution, as they may be moved to other platforms in a short time, when the physical server is experiencing increased overload.
- Backing up virtual machines and restoring them from backups takes much less time and is easier. In case of failure of equipment, the backup copy of the virtual server can be started immediately on another physical server.
- There is a simplicity and flexibility of managing a virtual infrastructure that enables centralized management of virtual servers and provides load balancing and migration.
- Decrease of demand for floor space and electric power and cooling, reducing electricity costs by 80 percent. [2]
- By simplifying the management of virtual servers entails savings on technical specialists, managing the infrastructure of the data center.

The university acquired 57 Hewlett-Packard servers with 12-core AMD Opteron 6174 processors and with a frequency of 2.2 GHz. The total number of processors is 132, and the total amount of installed RAM is 6912 GB. The two central modular Hewlett-Packard switches provide a communication with the servers combined by a proprietary technology for making backup called IRF at a speed of 10 GB Ethernet. (see the *Diagram 1* bellow)

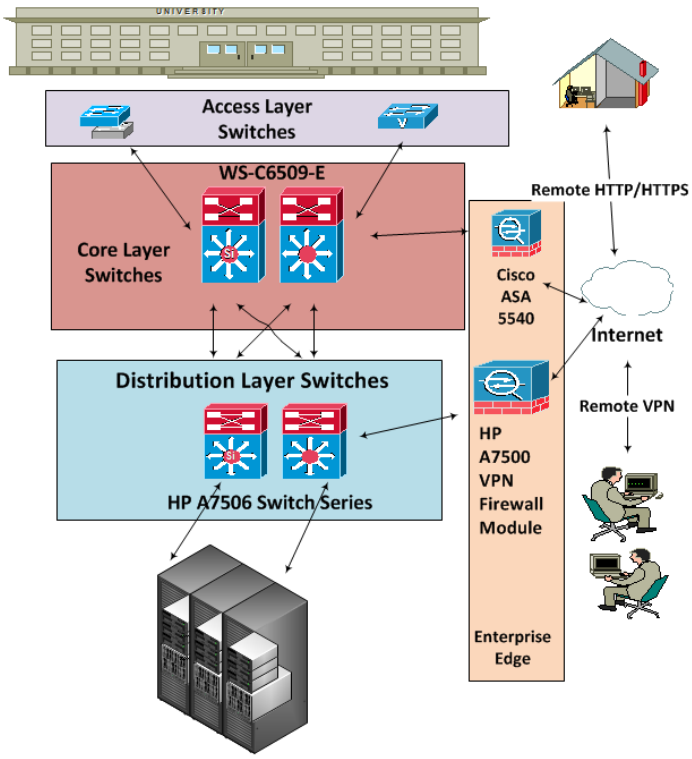


Diagram 1

SAN (fibre channel) network was also deployed at speeds up to 8Gb/ps. For storage, we used 3par array oriented on virtual software of 100 Tb size.

2.1 VMware-based virtual environment

The software package called VMware vSphere Enterprise Plus license was purchased and implemented as virtualization software, which is one of the pioneers of this technology, which provides great features, both today and in the future.[3] It was developed by a software company VMware.[4] In 1998, VMware has patented its technology virtualization software and has since released many effective and professional virtualization products at various levels. We have tested and tried using the freeware version of this software product of this company, before procuring this product.

The hypervisor Vmware ESXi version 4.1 was installed on each of the 57 servers. VMware first introduced its ESXi hypervisor at the end of 2007. [5] To provide fault tolerance all servers have been divided into two groups. The first group was dedicated for organizing business administrative services, and the second group was dedicated for services needed for teaching and for research centers. Both groups were combined into two fault-tolerant clusters for high availability with load balancing.

VMware vCenter Server, a critical component of a VMware-based virtual environment, is used to manage the entire infrastructure. [6] This solution allows reallocating computing resources between virtual servers depending on load and resource requirements.

In the event of failure of one or more servers in a cluster, a group of virtual machines that were located on the failed server is automatically restarted on the surviving server. A workload of each server is taken into account, when we restart virtual machines on a surviving server through dynamic allocation of resources. A clustering service called vCenter Server HeartBeat is deployed to ensure a high availability of the server management service VMware vCenter Server, which combines two servers Vmware Vcenter active and passive. These two constantly exchange messages about the availability and in the case of the failure of the active server - all the services will automatically start on the passive server. The vCenter Server HeartBeat works like a traditional cluster. [7] We need to have a private link between the Active/Passive nodes, to maintain the heartbeat process.[7] vCenter Server HeartBeat hides the passive node from the network using packet filtering, so it is not entirely like MSCS where we have separate IP address for each node and virtual IP to operate on. [7]

The established infrastructure has successfully passed several tests for fault tolerance, during which we emulated various emergency situations.

Up until this year the university experienced difficulties associated with maintenance of computer classes and access to students' personal computers. As a result of analysis, we have implemented desktop virtualization VDI (Virtual Desktop Infrastructure), which reduced the cost of supporting IT-infrastructure and provided students with the opportunity to work with all the necessary resources from any computer and thin client. [8]

When using this solution - instead of physical machines - tasks are performed on the virtual machines in a virtual environment, which are based on servers in the data center. A user can connect to a virtual machine, from any workstation and can work with it through the network. In this case a physical workstation serves as an access point, and is not tied to a particular place or user. This allows creating a virtual machine for each student without having to buy additional computers. Now we can use "thin clients" or regular PCs as an access point. This kind of setup has a high degree of security and virus protection, because of centralized storage of all the workstations on the servers of the university.

VMware View holds a central place among the proposed solutions of VMware, which was chosen for the task desktop virtualization. A new remote access protocol PCoIP (PC-over-IP) has been used to implement the technology of remote desktops, which is able to adapt to the characteristics of a network connection and the capabilities of a computer client, choosing the optimum parameters for the job. PCoIP was designed and developed by a company called Teradici. PCoIP is a type of display protocol used by remote desktops when doing desktop virtualization. [9] A display protocol is what delivers the desktop from the host server to the remote user along with capturing mice and keyboard inputs.[9] The protocol transmits HD-image access to USB devices, access to LAN and WAN. It served as the basis for implementing

virtualization platform VMware vSphere / ESXi.

We have deployed three pools of virtual desktops with their own unique settings. The first pool is designed for students, the second pool for faculty, and the third for administrative and academic management. This allowed granting access for users to a full-fledged workstations, running Windows 7 Professional, to meet the requirements of each class of users. For the infrastructure of VDI, we used a separate fail-proof cluster of VMware. The first two pools are used to allow access for students and teachers to personal workstations, and personal information. The third pool is used for temporary access under guest credentials in public access places and a library.

View Client was installed on a thin client or client system, which provides access to the workstations. Two versions are offered – the regular and «with local mode». The second one allows us to upload a virtual machine on our local computer to work without a connection to the View Manager.

A problem with a speed of PCoIP protocol was noticed when using VDI, resulting in noticeable slowdown of windows personal workstations. At the moment, we are looking for solutions to optimize the protocol speed together with VMware experts. This experience should be a good feature to research and to discuss in future papers.

3 Conclusions

In conclusion of this paper I would like to mention that VMware View is a complex product, but easy to manage. Most of the setup options are intuitive and transparent to our systems administrators and do not require relying on documentation very often. Overall, the implementation of virtualization technologies as part of the university's data center was successful. The selected technologies have proved to be efficient and functional, and the overall goals have been achieved.

4 References

[1] http://www.vmware.com/ru/virtualization/cost-savings/index.html

[2] http://www.vmware.com/ru/products/datacenter-virtualization/vsphere/

[3] VSphere Enterprise Plus: Persuading your boss to upgrade your VMware licenses, by Eric Siebert: http://searchvmware.techtarget.com/tip/VSphere-Enterprise-Plus-Persuading-your-boss-to-upgrade-your-VMware-licenses

[4] http://www.vmware.com/products/vsphere/mid-size-and-enterprise-business/buy.html

[5] ESX vs. ESXi: Convincing your boss to move to ESXi, by Eric Siebert: http://searchvmware.techtarget.com/tip/ESX-vs-ESXi-Convincing-your-boss-to-move-to-ESXi

[6] Stratus Uptime Appliance For VMware vCenter Server, by Dan Kusnetzky, February 22, 2012: http://www.zdnet.com/blog/virtualization/stratus-uptime-appliance-for-vmware-vcenter-server/4644

[7] vCenter Server Heartbeat, the concept, deployment and considerations, by Hany Michael, April 4th, 2009: http://www.hypervizor.com/2009/04/vcenter-server-heartbeat-the-concept-deployment-and-considerations/

[8] Virtual Desktop Infrastructure Has Cost-Cutting Potential, by Cameron Sturdevant, January 13 2009: http://www.eweek.com/c/a/IT-Infrastructure/Virtual-Desktop-Infrastructure-Has-CostCutting-Potential/

[9] PC-over-IP remote display technology: The inner workings of VMware View 4, by: Eric Siebert, November 11 2009: http://itknowledgeexchange.techtarget.com/virtualization-pro/pc-over-ip-remote-display-technology-the-inner-workings-of-vmware-view-4/

Online Task Scheduling Algorithm for Sporadic Tasks in Single-ISA Heterogeneous Multi-Core Architectures*

Yen-Ting Huang[1], Yi-Jung Chen[2], and Dyi-Rong Duh[3,†]
[1,2]*Department of Computer Science and Information Engineering*
National Chi Nan University
Puli, Nantou Hsien 54561, Taiwan
{s98321504, yjchen}@ncnu.edu.tw
[3]*Department of Computer Science and Information Engineering*
Hwa Hsia Institute of Technology
111, Gong Jhuan Rd., New Taipei City 23568, Taiwan
drduh@cc.hwh.edu.tw

***Abstract* –Single-ISA heterogeneous multi-core architecture has been proposed to reduce processor power dissipation. This work targets single-ISA heterogeneous multi-core architecture, and proposes an on-line task scheduling algorithm that dynamically assigns independent sporadic tasks in the system. Different from previous works that schedule tasks to minimize power consumption or maximize performance only, the proposed on-line task scheduling algorithm considers performance and power consumption at the same time. The experimental results show that, when comparing the energy-delay product, our algorithm has only 1.07 times over the algorithm that has oracle information and achieves the best results on the average.**

Keywords - Single-ISA multi-core architecture; heterogeneous multi-core architecture; performance; power consumption; on-line algorithm.

I. INTRODUCTION

Recently, single-chip multi-core processors have been a new trend in the design of high performance and low power consumption microprocessors [4, 5, 6, 7]. In order to provide significantly higher performance in the same area than a classic chip multiprocessor, a single instruction set architecture (single-ISA) heterogeneous multi-core architecture, which is a chip multiprocessor made up of cores of varying size, performance, and complexity, has been proposed by Kumar et al. [5]. In this architecture, the various tasks of a diverse workload are assigned to the various cores for reducing power consumption or improving performance according to the user demand. Notably, multi-core processors improve performance through thread-level parallelism but this type of architecture can provide high single-thread performance even when thread parallelism is low [5]. Since each core in single-ISA heterogeneous chip multiprocessor executes the same ISA, each task or task phase can be mapped to any of the cores [6]. The heterogeneous multi-core architecture possesses different type of cores and the multi-core architecture consists of the same type of cores. According to the different tasks, heterogeneous multi-core architecture can provide more suitable cores than homogeneous multi-core architecture. The energy consumption and the running time are different when different tasks work on different cores. Some cores provide high performance may consume more energy and some cores consume less energy may execute more slowly. Different tasks suit to different cores for lower power consumption and shorter time. If the tasks are always assigned on a high speed core, it may spend shorter time than assigned on a low energy core, but increase the power consumption. If the tasks are always assigned on a low energy core, it may spend a longer time than assign on a high speed power core, but reduce the power consumption. Therefore, the scheduling of tasks running on a single-ISA heterogeneous multi-core architecture is an important issue.

Several works that target single-ISA heterogeneous multi-core architecture have been proposed. Kumar et al. proposed the idea of single-ISA heterogeneous multi-core architecture to reduce processor power dissipation [4]. Rao et al. used the upper bound of the core's temperature to get the maximum throughput [7]. Then efficiently online computed the speed of the cores and thus got the speed matrix. Impose the matrix to determine the optimum speed combination. Xu et al. proposed a light-weight operating system scheduler for single-ISA asymmetric multi-core processor by using online profiling [8]. Xu et al. classified cores to fast cores and slow cores, analyzed the programs, assigned the CPU-intensive programs run on fast core and assigned the memory-intensive or I/O-intensive programs run on slow core. Notably, Kumar et al. only consider the offline problem. Since the coming time of every task is usually unknown, this work focuses on the more reality problem–online task scheduling problem. Recall that Rao et al.'s work should first computes the speed matrix such that the optimum speed combination can be obtained. Xu et al.'s work only classified the tasks to two types: CPU-intensive and I/O-intensive. Each task can be partitioned into several phases which may not work well on one core. This paper decomposes

* This work is partially supported by NSC 100-2221-E-146-014-.
† Correspondence to: D.-R. Duh; E-mail address: drduh@cc.hwh.edu.tw

every task into several phases, maps them to appropriate cores and analyzes every phase of the task.

This work derives an on-line task scheduling algorithm for sporadic tasks that execute in single-ISA heterogeneous multi-core architectures. We assume the tasks are invoked randomly, and the arrival time of each task in the task set is unknown. The goal of the proposed on-line task scheduling algorithm is to reduce the energy-delay product. The proposed algorithm is a greedy-based method that assigns a coming task to the core that has the least energy-delay product and among the unoccupied cores. We compare the proposed algorithm to three algorithms: the random algorithm, the energy-optimize algorithm and the oracle algorithm. When the tasks coming, the random algorithm assigns the tasks to the cores randomly; the energy-optimize algorithm assigns the tasks to the core which energy consumption is the least in order, and the oracle algorithm will compute the energy-delay products of all sets, and choose the set which has least energy-delay product. In energy-optimize algorithm, only one task can run on a core at a time, the task will always choose the core which saving energy most. This algorithm saves most energy but wastes time. The oracle algorithm is an algorithm that we assume we already know the coming time of the tasks and always choose the most saving energy set of various tasks on various cores.

The experimental result shows that the average of energy-delay products of our algorithm have only 1.07 times over the oracle algorithm, which is very close to the oracle algorithm. Moreover, the energy consumption of the proposed on-line task scheduling algorithm achieves 1.2 times over the energy-optimize algorithm on the average.

The rest of the paper is organized as follows. Section 2 describes the proposed online task scheduling algorithms on the single-ISA heterogeneous multi-core architecture. Section 3 states the experimental setup this paper. Conclusion and summary of this work are drawn in Section 4.

II. Proposed Algorithm

A greedy-based online task scheduling algorithm is provided to reduce the energy-delay product. Assume energy consumption and execution time of a task running on a specific core is known in advance. Each single core could only run a single task at a time, and the unused cores are completely powered down. The tasks will be reassigned to cores when a context switch happens. Tasks are non-preemptive; every task will not be interrupt if the task is starting executing.

The input of the online task scheduling algorithm is the target single-ISA heterogeneous platform and task set. The platform includes the energy consumption and working frequency of each core. For each task in the task set, we record the energy consumption and running time of it on each single-ISA core. The output of the online task scheduling algorithm is a scheduling of tasks to cores. The optimization goal of this work is to reduce the energy-delay product of the tasks on the cores. The energy-delay product is the total execution time multiplied by the sum of the energy consumption of each task. Let T and E_i denote total execution time and energy consumption of task i. Show the formula as follow.

energy-delay product = $T \times \Sigma E_i$

As mentioned earlier, the proposed on-line task scheduling algorithm is a greedy-based method that assigns tasks to the cores. The flow of the online task scheduling algorithm is detailed as follows and shown in Figure 1.

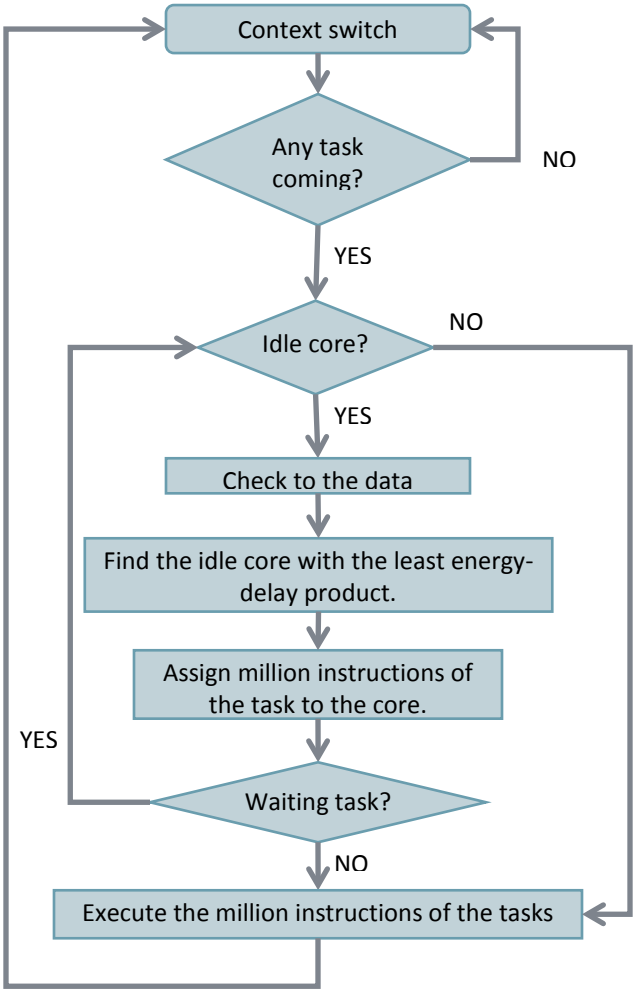


Figure 1. The flow of the online task scheduling algorithm.

Algorithm Online Task Scheduling

{

When the context switch happens, check if there any task coming.

If there is any task coming, check if there is any idle core.

1. If so, check to the information of energy consumption and execution time of a task on each single-ISA core.
2. Find the idle core with the least energy-delay product and then assign one million instructions of the tasks.
3. Check if there is any waiting task.
   If so, check if there is any idle core.

If so, jump to 1.

4. Otherwise, execute the million instructions of the tasks and wait for the next context switch happens.

Otherwise, wait for the next context switch happens.

} // End of **Algorithm Online Task Scheduling**

III. Experimental Result

A. Experimental Setup

To evaluate the proposed on-line task scheduling algorithm, we develop a simulation platform by C++ to simulate the behavior of the proposed algorithm on the target single-ISA multi-core architecture. The host machine to run the simulation platform is a PC with Intel® Core™ i5-2005S 2.7GHz CPU, 4.00GB RAM with Windows 7.

In our experiments, we assume the target single-ISA heterogeneous multi-core architecture is composed of four cores (Core A, Core B, Core C and Core D). The ranges of the power of four cores refer to the cores of the Alpha [1, 2, 3], they are Alpha 21064 (known as the EV4), Alpha 21164 (EV5), Alpha 21064 (EV6) and Alpha 21364A (EV78). Table 1 is the information of the cores of the Alpha this paper refers to. The range of the power of Core A refers to Alpha 21064. The range of the power of Core B refers to Alpha 21164. The range of the power of Core C refers to Alpha 21264 and the range of the power of Core D refers to Alpha 21364A. Therefore the ranges of the power consumption of each of the four cores are listed as follows: Core A: 20~40 Watt, Core B: 46~66 Watt, Core C: 63~83 Watt and Core D: 110~130 Watt. The range of the execution time is usually related to the power of core in an inverse proportion. Therefore we set the range of the execution time of each core based on this assumption. As shown in Table 2, the ranges of the execution time of each of the four cores are listed as follows, Core A: 110~130 unit time, Core B: 63~83 unit time, Core C: 46~66 unit time and Core D: 20~40 unit time.

Table 1. The information of the cores this paper refers to.

Core	Model number	Power [W]	Die size [mm^2]
EV4	21064	30	234
EV5	21164	56	299
EV6	21264	73	314
EV78	21364A	120	300

Table 2. The information of the cores.

Core	Range of the Power [W]	Range of the Execution Time [unit time]
Core A	20~40	110~130
Core B	46~66	63~83
Core C	63~83	46~66
Core D	110~130	20~40

For every task set, assume there are ten distinct tasks; each task has ten million instructions, and every core randomly picks a power and an execution time in the range of the core. The ten distinct tasks will also come randomly. The context switch time is defined as the interval of one million instructions (corresponding roughly to an OS time-slice interval).

B. Algorithm of Comparison

This section compares the online task scheduling algorithm to three algorithms: the random algorithm, the energy-optimize algorithm and the oracle algorithm.

The first algorithm is the random algorithm. When the context switch happens, each coming task will be randomly assigned to any idle core until all cores are occupied. If all cores are occupied, the waiting task will wait until any core is free.

The second algorithm is the energy-optimize algorithm. It is the optimum algorithm to save the energy. Every coming or running task will choose the core with the lowest energy consumption by every millions instructions. Only one task can be executed at a time. The next coming task will wait for the previous task finished. Next coming task then choose the core with the lowest energy consumption. This algorithm might spend lots of time but the tasks can always choose the most energy saving core.

The third algorithm is the oracle algorithm. This algorithm provides the upper bound of the optimization goal. Assume the execution times of the tasks are known. The coming task will choose the core that has the least energy-delay product. If there are other tasks need to be executed, calculate the information of the tasks on the cores including the execution time, the power consumption and the energy-delay product. According to the information of the tasks to the cores, choose the least of the sum of the energy-delay product combination. Notice that if all cores are occupied, the next task will wait for the previous tasks until there is an idle core.

C. Simulation Result

This section analyzes the synthesis results of the online task scheduling algorithm. This work runs one hundred task sets for the simulator, and gets one set of the energy consumptions and running times of each task set on each single-ISA core. Then calculate their energy-delay product.

Figure 2 shows the results. The energy-delay product is normalized to the oracle algorithm. In 100 different task sets of experiments, in few of the sets, the energy-delay product of the random algorithm is close to the online task scheduling algorithm. Significantly, for most of the sets, the energy-delay products of the online task scheduling algorithm are much smaller than the energy-delay product of the random algorithm. For the information shown in Figure 2, the online task scheduling algorithm is close to the oracle algorithm but the online task scheduling algorithm do not know the coming time of the tasks.

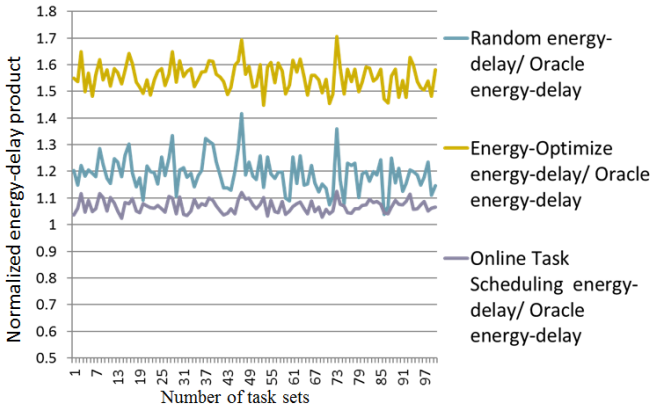


Figure 2. Normalized the energy-delay product to the oracle algorithm.

This paper also aims at the tasks if they consume more than 20% of energy to the original tasks. Figure 3 is the energy-delay product normalized to the oracle algorithm when the tasks spend more than 20% of energy. According to Figure 3, for few sets, the energy-delay product of the random algorithm is close to the online task scheduling algorithm and even smaller than the online task scheduling algorithm, but for the average, the energy-delay product of online task scheduling algorithm still much better than the random algorithm. The energy-delay product of online task scheduling algorithm is still close to the oracle algorithm.

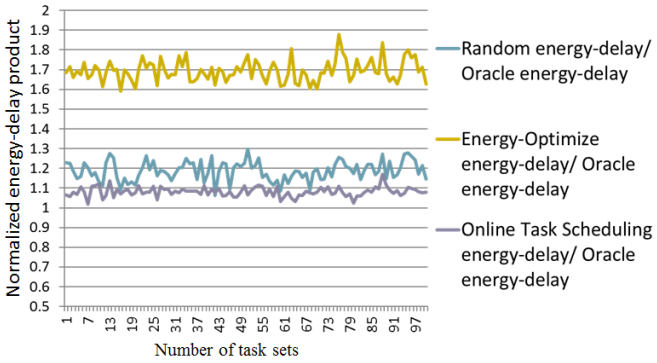


Figure 3. Normalized the energy-delay product to the oracle algorithm when the tasks spend more than 20% of energy.

Figure 4 is the energy consumption normalized to the energy-optimize algorithm. The random algorithm spends more energy than the online task scheduling algorithm and the oracle algorithm. The energy consumption of the online task scheduling is close to the energy consumption of the oracle algorithm.

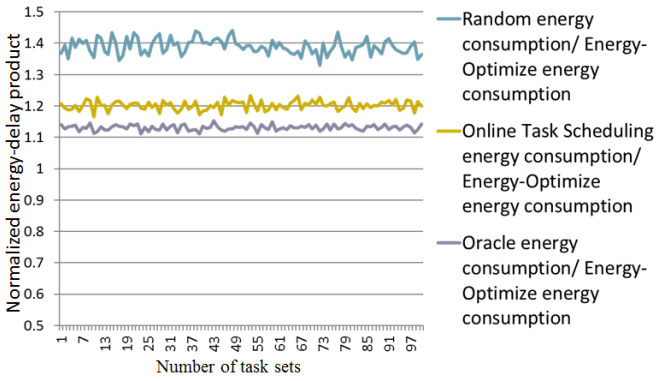


Figure 4. Normalized the energy consumption to the energy-optimize algorithm.

IV. Conclusion

This work provides an online task scheduling algorithm to reduce the energy-delay product of the single-ISA heterogeneous multi-core architecture when the coming times of the tasks are unknown. The energy-delay product of the online task scheduling algorithm is close to the oracle algorithm and better than the random algorithm and the energy-optimize algorithm. Furthermore, the energy consumption of the online task scheduling algorithm is close to the oracle algorithm and better than the random algorithm. According to the requirement, user can choose the most suitable algorithm.

References

[1] Alpha 21064 and Alpha 21064A: Hardware Reference Manual.

[2] Alpha 21164 Microprocessor: Hardware Reference Manual.

[3] Alpha 21264/EV6 Microprocessor: Hardware Reference Manual. Compaq Corporation, 1998.

[4] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan and D. M. Tullsen, "Single-ISA Heterogeneous Multi-core Architecture: The Potential for Processor Power Reduction", In: *Proceeding of the 36th International Symposium on Microarchitecture*, Dec. 2003, pp. 81–92.

[5] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi and K. I. Farkas, "Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance", In: *Proceedings of the 31st International Symposium on Computer Architecture*, June, 2004, pp. 64–75.

[6] R. Kumar, D. M. Tullsen, N. P. Jouppi and P. Ranganathan, "Heterogeneous Multi-core Architectures ", *IEEE Transactions on Computers*, pp. 32–38, 2005.

[7] R. Rao and S. Vrudhula, "Efficient Online Computation of Core Speeds to Maximize the Throughput of Thermally Constrained Multi-Core Processors." In: *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Jose, California, pp. 537–542, November 10–13, 2008.

[8] Y. Xu, L. Du and Z. Zhang, "A Light-Weight Scheduler for Single-ISA Asymmetric Multi-core Processor Using Online Profiling." In: *Proceedings of 12th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 2011, pp. 183–188.