i-Core: A run-time adaptive processor for embedded multi-core systems

Jörg Henkel, Lars Bauer, Michael Hübner, and Artjom Grudnitsky
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{henkel, lars.bauer, michael.huebner, artjom.grudnitsky} @ kit.edu
Submitted as Invited Paper to ERSA 2011

Abstract

We present the i-Core (*Invasive Core*), an Application Specific Instruction Set Processor (ASIP) with a run-time adaptive instruction set. Its adaptivity is controlled by the run-time system with respect to application properties that may vary during run-time. A reconfigurable fabric hosts the adaptive part of the instruction set whereas the rest of the instruction set is fixed. We show how the i-Core is integrated into an embedded multi-core system and that it is particularly advantageous in multi-tasking scenarios, where it performs applications-specific as well as system-specific tasks.

1. Introduction and Motivation

Embedded processors are the key in rapidly growing application fields ranging from automotive to personal mobile communication, computation, and entertainment, to name just a few. In the early 1990s, the term ASIP has emerged denoting processors with an application-specific instruction set (Application Specific Instruction-set Processors). They are more efficient in one or more design criteria like 'performance per area' and 'performance per power' [1] compared to mainstream processors and eventually make today's embedded devices (which are often mobile) possible. Nowadays, the term ASIP comprises a far larger variety of embedded processors allowing for customization in various ways including a) instruction set extensions, b) parameterization and c) inclusion/exclusion of predefined blocks tailored to specific applications (like, for example, an MPEG-4 decoder) [1]. An overview for the benefits and challenges of ASIPs is given in [1-3].

A generic design flow of an embedded processor can be described as follows:

i) an application is analyzed/profiled
ii) an instruction set extension (containing so-called *Special Instructions*, SIs) is defined
iii) the instruction set extension is synthesized together with the core instruction set
iv) retargetable tools for compilation, instruction set simulation, and so on, are (often automatically) created and application characteristics are analyzed
v) the process might be iterated several times until design constraints comply

Automatically detecting and generating SIs from the application code (like in [4]) plays a major role for speeding-up an application and/or for power efficiency. Profiling and pattern matching methods [5, 6] are typically used along with libraries of reusable functions [7] to generate SIs.

However, fixing critical design decisions during design time may lead to embedded processors that can hardly react to an often non-predictive behavior of today's complex applications. This does not only result in reduced efficiency but it also leads to an unsatisfactory behavior when it comes to design criteria like 'performance' and 'power consumption'. A means to address this dilemma is reconfigurable computing [8-11] since its resources may be utilized in a time-multiplexed manner (i.e. reconfigured over time). A large body of research has been conducted in interfacing reconfigurable computing fabrics with standard processor cores (e.g. using an embedded FPGA [12-14]).

This paper presents the i-Core, a reconfigurable processor that provides a high degree of adaptivity at the level of instruction set architecture (through using reconfigurable SIs) and microarchitecture (e.g. reconfigurable caches and branch predictions). The potential performance advantages at both levels are exploited and combined which allows for a high degree of adaptivity that is especially beneficial in run-time varying multi-tasking scenarios.

Paper structure: Section 2 presents state-of-the-art related work for reconfigurable processors. An overview of our i-Core processor is given in Section 3 where we explain the way it is integrated into a heterogeneous multi-core system and the kind of adaptivity it provides with respect to SIs and the microarchitecture. Section 4 explains how SIs are modeled and how the programmer can express which SIs are demanded by the application to trigger their reconfiguration. Performance results of applications executing on the i-Core are given in Section 5 and conclusions are drawn in Section 6.

2. Related Work

Diverse approaches for reconfigurable processors were investigated particularly within the last decade [8-11]. The Molen Processor couples a reconfigurable coprocessor to a core processor via a dual-port register file and an arbiter for shared memory [15]. The application binary is extended to include instructions that trigger the reconfigurations and control the usage of the reconfigurable coprocessor. The OneChip project [16, 17] uses tightly-coupled Reconfigurable Functional Units (RFUs) to utilize reconfigurable computing in a processor. As their speedup is mainly ob-

tained from streaming applications, they allow their RFUs to access the main memory, while the core processor (i.e. the non-reconfigurable part of the processor) continues executing [18]. Both approaches target a single-tasking environment and statically predetermine which SIs shall be reconfigured at 'which time' and to 'which location' on the reconfigurable fabric. This will lead to conflicts if multiple tasks compete for the reconfigurable fabric (not addresses by these approaches).

The Warp Processor [19] automatically detects computational kernels while the application executes. Then, custom logic for SIs is generated at run-time through on-chip micro-CAD tools and the binary of the executing program is patched to execute them. This potentially allows adapting to changing multi-tasking scenarios. However, the required online synthesis may incur a non-negligible overhead and therefore the authors concentrate on scenarios where one application is executing for a rather long time without significant variation of the execution pattern. In these scenarios, only one online synthesis is required (i.e. when the application starts executing) and thus the initial performance degradation accumulates over time. Adaptation to frequently changing requirements –as typically demanded in a multi-tasking system– is not addressed by this approach.

The Proteus Reconfigurable Processor [20] extends a core processor with a tightly-coupled reconfigurable fabric. It concentrates on Operating System (OS) support with respect to SI opcode management to allow different tasks to share the same SI implementations. Proteus' reconfigurable fabric is divided into multiple Programmable Functional Units (PFUs) where each PFU may be reconfigured to contain one SI (unlike ReconOS [21], where the reconfigurable hardware is deployed to implement entire threads). However, when multiple tasks exhibit dissimilar processing characteristics, a task may not obtain a sufficient number of PFUs to execute all SIs in hardware. Therefore, some SIs will execute in software, resulting in steep performance degradation.

The RISPP processor [22, 23] uses the reconfigurable fabric in a more flexible way by introducing a new concept of SIs in conjunction with a run-time system to support them. Each SI exists in multiple implementation alternatives, reaching from a pure software implementation (i.e. without using the reconfigurable fabric) to various hardware implementations (providing different trade-offs between the amount of required hardware and the achieved performance). The main idea of this concept is to partition SIs into elementary reconfigurable data paths that are connected to implement an SI. A run-time system then dynamically chooses one alternative out of the provided options for SI implementations, depending on run-time application requirements. It focuses on single-tasking scenarios and does not aim to share the reconfigurable fabric among multiple tasks or among user tasks and OS tasks. KAHRISMA [24, 25] extends the concepts of RISPP by providing a fine-grained reconfigurable fabric along with a coarse-grained reconfigurable fabric that can then be used to implement SIs and to realize pipeline- or VLIW processors. Therefore, KAHRISMA supports simultaneous multi-tasking (one task per core), but it does not consider executing multiple tasks per core or adapting the microarchitecture (e.g. cache- or branch-prediction) of a core.

Altogether, only Proteus explicitly targets multi-tasking systems in the scope of reconfigurable processors that use a fine-grained reconfigurable fabric. The concept of PFUs does not provide the demanded flexibility to support multiple tasks efficiently though. RISPP and KAHRISMA provide a flexible SI concept but the challenge of sharing the reconfigurable fabric and the configuration of the microarchitecture among competing tasks is not addressed. The Warp processor provides the potentially highest flexibility, but it comes at the cost of online synthesis, which limits the scenarios in which this flexibility can be efficiently used. Hence, when studying state-of-the-art approaches, the following challenge remains: providing an adaptive reconfigurable processor that can share the reconfigurable fabric efficiently among multiple user tasks while providing an adaptive microarchitecture that can adapt to varying task requirements.

3. i-Core Overview

The i-Core is a reconfigurable processor that provides a run-time reconfigurable instruction set architecture (ISA) along with a run-time reconfigurable microarchitecture. The ISA consists of two parts, the so-called core ISA (cISA) and the Instruction Set Extension (ISE). The cISA is statically available (i.e. implemented with non-reconfigurable hardware) and the ISE represents the task-specific components of the ISA that are realized as reconfigurable Special Instructions (SIs). The i-Core uses a fine-grained reconfigurable fabric (i.e. an embedded FPGA, e.g. [12-14]) to provide

i) task-specific ISEs,
ii) OS-specific ISE, and
iii) an adaptive microarchitecture that – among others – allows for supporting and executing both kinds of ISEs efficiently by performing run-time reconfigurations

This approach exceeds the concept of state-of-the-art ASIPs, as it adds flexibility and additionally enables dynamic run-time adaptation towards the executing application to increase the performance.

The reconfigurable microarchitecture characterizes the concrete processor-internal realization of the i-Core for a given ISA. It refers to the internal process of instruction handling and it is developed with respect to a predefined ISA (SPARC-V8 [26] in our case). Summarizing, the ISA specifies the instruction set that can be used to program the processor (without specifying how the instructions are implemented) and the microarchitecture specifies its implementation and further ISA-independent components (e.g. caches and branch prediction).

Figure 1 shows how the i-Core is embedded into a heterogeneous loosely-coupled multi-core system which is

partitioned into tiles that consist of processor cores and local shared memory. Each tile is connected to one router that is part of an on-chip network to connect the tiles to each other and to external memory. Within one tile, none, one, or multiple i-Core instances are located and connected to the tile-internal communication structure (similar to the other CPUs of the tile). The CPUs within the tile access the shared memory (an application-managed scratchpad) via the local bus. The i-Core uses a dedicated connection to the local memory, using two 128-bit ports to provide a high memory bandwidth to expedite the execution of SIs. When multiple i-Cores are situated within one tile, then the access to the local memory and the reconfigurable fabric is shared among them.

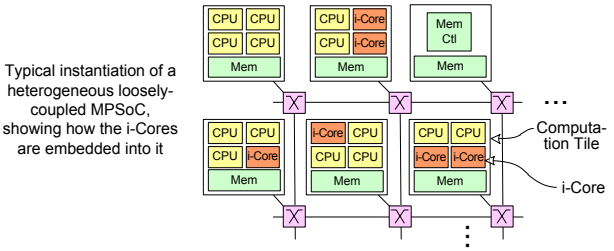

Figure 1: Integration of the i-Core into a heterogeneous multi-core system with on-chip interconnect network

Figure 2 provides an overview of the i-Core-internal adaptation options of the ISE and the microarchitecture. The ISE provides task-specific and system-specific SIs. Whereas the task-specific SIs are particularly targeted towards a certain application or application domain, the system-specific SIs support basic operating system (OS) functionalities that are required to execute tasks and to manage the multi-core system. The OS can (to some degree) be viewed as a set of tasks (e.g. task mapping, task scheduling, etc.) that can be accelerated by SIs. Typically, it is impossible to fulfill all ISE requests due to the limited size of the reconfigurable fabric. The SIs that are not implemented on the reconfigurable fabric at a certain point in time can be executed through an 'unimplemented instruction' trap as presented in more detail in [27].

SIs are prepared at compile time of the tasks and the calculations that are performed by an SI are fixed at run time. Instead, the *implementation* of a particular SI may change during run-time (details are explained in Section 4.1). These adaptations correspond to ISE-specific adaptations at the microarchitecture level. Figure 2 shows a reconfigurable fabric – in addition to the hardware of the processor pipeline – that is employed to realize an ISE. Depending on the executing tasks and their specific ISE requirements, the reconfigurable fabric is allocated or – as we call it – *invaded* to realize a certain subset of the requested ISEs (that is why we call it an *invasive Core*, i.e. i-Core). The concepts of *invasive computing* [28] are used to manage the competing requests of different tasks. In the scope of reconfigurable processors this means that each task specifies the SIs that it uses (i.e. 'its requests'; details are presented in Section 4.2). Additionally, each task provides information which performance improvement (speedup) can be expected, depending on the size of the reconfigurable fabric that is assigned to it. A run-time system (part of the OS) then decides for the tasks that compete for the resources, which task obtains which share of the reconfigurable fabric (similar to the approach presented in [29] where the fabric of one task is partitioned among the SIs of that task).

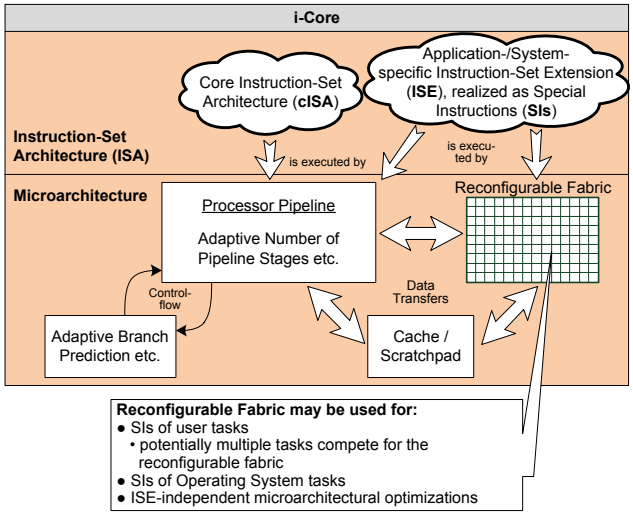

Figure 2: The Instruction-Set Architecture and Microarchitecture Adaptations of our i-Core

The highly adaptive nature of modern multi-tasking embedded systems intensifies the potential advantages that come along with an adaptive microarchitecture and instruction set architecture. The adaptations of the instruction set comprise flexible task-specific SIs that support at run-time adaptations in terms "performance per area" (details given in Section 4.1) depending on the number of tasks that execute on an i-Core at a specific time (and thus the amount of reconfigurable fabric that is available per task). In addition to the ISE, the i-Core also supports adapting its microarchitecture. The microarchitecture adaptations are independent upon the instruction set and they comprise:

- adaptive number of pipeline stages
- adaptive branch prediction
- adaptive cache/scratchpad configuration

Even though these optimizations are ISA-independent, they may significantly increase the performance of the executing task (or reduce the power consumption etc.). The number of pipeline stages is altered by combining neighbored stages and bypassing the pipeline registers between them. This reduces the maximal frequency of the processor but it may lead to power savings (reduced number of registers) and may be beneficial in terms of performance for applications where a complex control flow leads to many branch miss-predictions. Additionally, techniques like pipeline balancing [30] or pipeline gating [31] can be applied. Depending on a task's requirements, the branch prediction scheme can also be changed dynamically, e.g. by providing

different schemes and letting the task decide which one to use. Another example of performance-oriented adaptive design is *branch history length adaptation*, e.g. Juan et al. [32] explore dynamic history-length fitting and develop a method for dynamically selecting a history length that accommodates the current workload.

In addition, the cache can be changed in various ways. For instance, Albonesi [33] proposes to disable cache ways dynamically to reduce dynamic energy dissipation. Kaxiras et al. [34] reduce leakage power by invalidating and turning off the cache lines when they hold data that is not likely to be reused. The approaches of [35-37] use an adaptive strategy to adjust the cache line size dynamically to an application. In addition to these approaches, the microarchitecture of the i-Core exploits the availability of the fine-grained reconfigurable fabric to extend the size and associativity of the cache. For example, the size of the cache (e.g. number of cache lines) can be extended (using the logic cells of the reconfigurable fabric as fast memory), further parallel comparators can be realized to increase the associativity of the cache, or additional control bits can be assigned to each cache line for protocol purpose (e.g. error detection/correction schemes). Additionally, the memory of the cache can be reconfigured to be used as a task-managed scratchpad memory.

In summary, instruction set and microarchitecture adaptations target task-specific optimizations, for example, a particular task might benefit from a certain SI (part of the ISA) and a certain branch prediction (part of the microarchitecture). Additionally, a particular task might also benefit from different ISA/microarchitecture implementations at different phases of its execution (e.g. different computational kernels), i.e. the requirements of a sole task may change over time. Depending on the tasks that execute at a certain time and their requirements, the adaptations focus on:

- some selected tasks (beneficial for those tasks at the cost of other tasks)
- operating system optimization (beneficial for all tasks)
- a trade-off between both

Determining this trade-off depends on the user-priorities of the executing tasks. This large degree of flexibility is an advantage in comparison to state-of-the-art adaptive processors (e.g. RISPP [22, 23] or KAHRISMA [24, 25]) as they focus on accelerating either the tasks or the operating systems (but not both) by improving either the instruction set or the microarchitecture (and gain, not both).

3.1. Partitioning the Reconfigurable Fabric among Special Instructions and Microarchitecture

The core ISA (cISA) is executed by means of a specific hardware at the microarchitecture level. Depending on the requirements of the executing task, the microarchitecture implementation of the cISA can be changed during runtime. For instance, a 5-stage pipeline implementation can be replaced by a faster 7-stage pipeline implementation as explained in Section 3.

Figure 3 illustrates an example for the different levels of adaptivity, using a task execution scenario in a sequence from a) to d). It shows how the execution pipeline, the cache, and the reconfigurable fabric may be *invaded* by different tasks (i.e. the resources are reconfigured towards the requirements of the task as explained in Section 3). Part a) of Figure 3 illustrates that the reconfigurable fabric can be used to accelerate OS functionality. This is especially beneficial, as the workload of the OS heavily depends on the behavior of the tasks, that is, 'when' and 'how many' system calls etc. will be executed. Therefore, providing static accelerators for the OS is not necessarily beneficial. Instead, the hardware may be reconfigured to accelerate other tasks in case the OS does not benefit from it at a certain time, as shown in part b) of the figure.

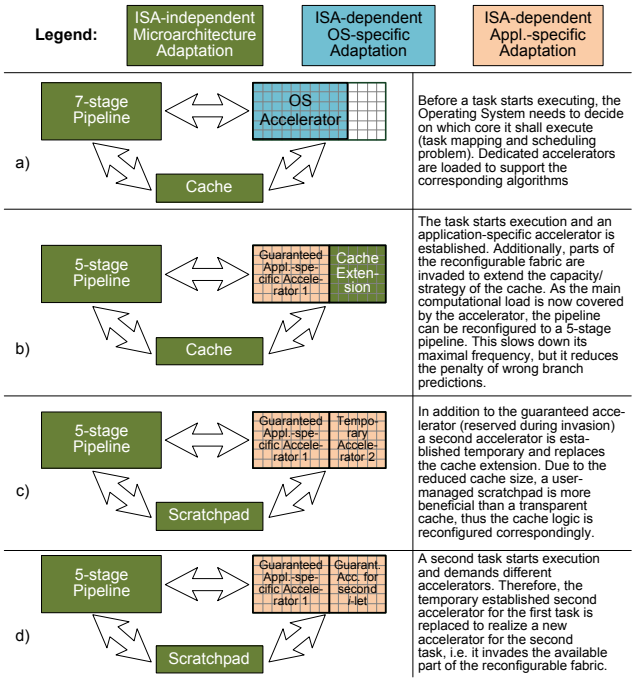

Figure 3: An example to demonstrate the adaptivity of the i-Cores, comprising ISA-independent adaptations as well as task-specific and OS-specific adaptations

Microarchitectural components like the cache and the processor pipeline can be adapted to different task requirements. Depending on the demanded time for performing these reconfigurations, the changes may be performed specific to the currently executing task or they may be performed for the set of tasks that execute on the i-Core. For instance, reconfiguring a 7-stage pipeline into a 5-stage pipeline only demands a few cycles and thus it can be performed as part of the task switch (in a preemptive multitasking system) when changing from one task to another. Instead, the reconfiguration time of the reconfigurable fabric is typically larger and changing the configuration as part of the task switch would increase the task switching time

significantly. Therefore, to share the reconfigurable fabric among multiple tasks, it needs to be partitioned dynamically as indicated in parts c) to d) of Figure 3. Therefore, at different points in time, the temporary allocation 'which part' of the reconfigurable fabric accelerates 'which task' changes dynamically. Parts c) and d) show that a task can obtain a guaranteed share of the reconfigurable fabric, but it may use a larger share of it temporarily.

4. Modeling and Using Reconfigurable Special Instructions

4.1. Special Instruction Model

As motivated in Section 3, the reconfigurable fabric is shared among several tasks and thus, the ISE of a particular task has to cope with an at compile-time unknown size of the reconfigurable fabric. The approach of so-called *modular SIs* allows for providing different trade-offs between the amount of required hardware and the achieved performance by breaking SIs into elementary reconfigurable data paths (DPs) that are connected to implement an SI. It was originally developed in the scope of the RISPP project [22] and is meanwhile integrated and extended in other projects as well (e.g. KAHRIMSA [24]). The basic idea of modular SIs is illustrated in Figure 4 (description and an example follows) and is based on a hierarchical approach that – among others– allows to represent that

a) an SI may be realized by multiple SI implementations, typically differing in their hardware requirements and their performance (and/or other metrics), and
b) a DP is not dedicated to a specific SI, but it can be used as a part of different SIs instead.

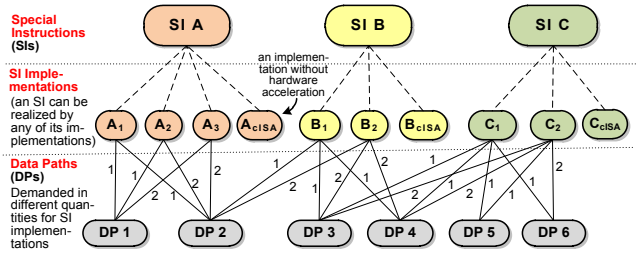

Figure 4: Hierarchical composition of SIs: multiple implementation alternatives exist per SI and demand data paths for realization; Figure based on [22]

Modular SIs are composed of DPs, where typically multiple DPs are combined to implement an SI. Figure 5 shows the Transform DP and the SAV (Sum of Absolute Values) DP to indicate their typical complexity. DPs are created and synthesized during compile time and they are reconfigured at run time. Technically, DPs are the smallest components that are reconfigured in modular SIs. An SI is composed of multiple DPs, as indicated in Figure 4 and illustrated with an example in Figure 5 and an SI implementation incorporates them in different quantities.

Figure 4 shows the hierarchy of SIs, SI Implementations, and DPs. At each point in time, a particular SI is implemented by one specific SI Implementation. It is noticeable that DPs can be used by different implementations of the same SIs and even by different SIs. This means that a DP can be shared among different SIs.

Each SI has one special Implementation that does not demand any DPs. This means that this SI implementation is not accelerated by hardware. If the SI shall execute but an insufficient amount of DPs needed to implement any of the other Implementations is available (i.e. reconfigured to the reconfigurable fabric), then the processor raises an 'unimplemented instruction' trap and the corresponding trap handler is used to implement the SI's functionality [27]. The trap handler uses the core Instruction Set Architecture (cISA) of the processor. Therefore, this cISA implementation allows bridging the reconfiguration time (in the range of 1 to 10 ms), i.e. the time until the required DPs are reconfigured.

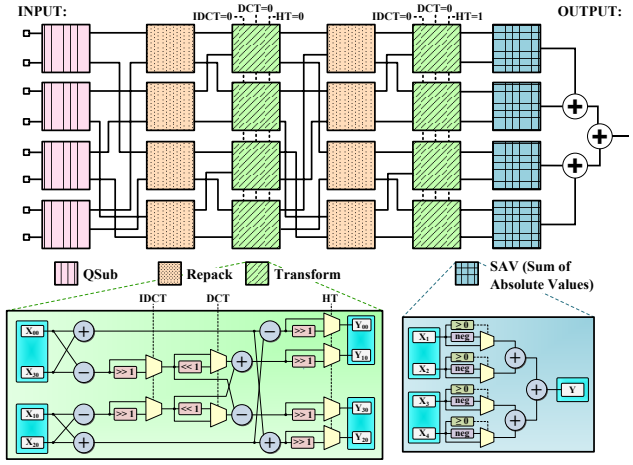

Figure 5: Example for the modular Special Instruction SATD (Sum of Absolute (Hadamard-) Transformed Differences); Figure based on [22, 27]

Figure 5 shows an example for a modular SI that is composed of four different types of DPs and that implements the Sum of Absolute Hadamard-Transformed Differences (SATD) functionality, as it is used in the Motion Estimation process of an H.264 video encoder. The data-flow graph shown in Figure 5 describes the general SI structure and each implementation of this SI has to specify how many *DP instances* of the four utilized *DP types* (i.e. QSub, Repack, Transform, and SAV) shall be used for implementation. Depending on the provided amount of DPs, the SI can execute in a more or less parallel way. When more DPs are available, then a faster execution is possible (that corresponds to a different implementation of same SI). Dynamically changing between these different performance-levels of an SI implementation allows reacting on changing application requirements or changing availability of the reconfigurable fabric.

4.2. Programming Interface for using the reconfigurable fabric

A task uses DPs on the reconfigurable fabric without the knowledge of low-level details such as partitioning the fab-

ric for different SIs, determining the sequence of 'which' DPs are loaded 'where' on the fabric, etc. These steps are handled by the run-time system of the i-Core. Nevertheless, the programmer needs to trigger these steps by issuing system calls (presented in this section). The code fragment in Figure 6 (a high-level excerpt from the main loop of a H.264 video encoder) illustrates the programming model of the i-Core.

Microarchitectural features are adapted using the *set_i_Core_parameter* system call. It ensures that any preconditions for an i-Core adaptation are met (e.g. emptying the pipeline before changing pipeline length, invalidating cache-lines before modifying cache-management parameters, etc.) and it then performs the adaptations. For example, in Line 3 of Figure 6, the i-Core pipeline length is set to 5 stages and branch prediction is switched to a (2, 2) correlating branch predictor.

To request a share of the reconfigurable fabric, the task issues the *invade* system call (Line 5). *Invade* selects a share of a resource and grants it to the task. There, the resource is the entire reconfigurable fabric of the i-Core, a part of which is assigned to the task. The size of the assigned fabric depends on the speedup that the application is expected to exhibit. Generally, the more fabric is available to the task, the higher the speedup, but the expected speedup for a given amount of fabric is task-specific. This 'speedup per size of assigned fabric' relationship is explored during offline profiling and passed as the *trade_off_curve* parameter to the *invade* system call. During execution of *invade*, the run-time system will use the trade-off curve to decide which share of the fabric will be granted to the task. In the worst case, the application will receive no fabric at all (due to e.g. all fabric being occupied by higher priority tasks), then all SIs need their core ISA implementation (see Section 4.1) for execution The share of the fabric assigned to an application corresponds to the *my_fabric* return value in Line 5. Requesting a share of the fabric is typically done before the actual computational kernels start.

Next, the implementations of the SIs that will be used during the next kernel execution (the 'while' loop in the code example, lines 6-15) must be determined. The application programmer does not need to know which SI implementations are available at run time, as long as it is guaranteed that the SI functionality is performed (the SI implementation determines the performance of the SI, but not its functionality). The application informs the run-time system which SIs it will use during the next kernel execution and the run-time system selects implementations for these SIs. This is accomplished by means of the *invade* system call again (Line 7). Here, the invaded resource is the application's own share of the reconfigurable fabric acquired earlier, which needs to be partitioned such that SI implementations for the requested SIs (SATD and SAD in the code example) fit onto it [29]. The programmer may also explicitly request specific DPs that shall be loaded into the fabric and the run-time system will consider these requests when selecting SI implementations for the task. This manual intervention may be used if the SI requirements of a kernel are rather static, i.e. the run-time system does not need to provide adaptivity for its implementation. Additionally, it can be used to limit the search for SI implementations and thus reduce the overhead of the run-time system.

```
1.  H264_encoder() {
2.      // Set i-Core microarchitecture parameters
3.      set_i_Core_parameter(pipeline_length=5,
            branch_prediction=2_2_correlation_predictor);
4.      // Invade a share of the reconfigurable fabric
5.      my_fabric=invade(resource=reconf_fabric,
                    performance=trade_off_curve);
6.      while (frame=videoInput.getNextFrame()) {
7.          SI_implementations=invade(resource=my_fabric,
                SI={SAD, trade_off_curve[SAD],
                        execution_prediction[SAD_ME]},
                SI={SATD, trade-off_curve[SATD],
                        execution_prediction[SATD_ME]} );
8.          infect(resource=my_fabric, SI_implementations);
9.          motion_estimation(frame, ...);
10.         ...
11.         SI_implementations=invade(resource=my_fabric, ...);
12.         infect(resource=my_fabric, SI_implementations);
13.         encoding_engine(frame, ...);
14.         ...
15.     }
16. }
```

Figure 6: Pseudo code example for invading the reconfigurable fabric and the microarchitecture

After the run-time system has decided which SI implementations to use, the application can start loading the required DPs by issuing the *infect* system call (Line 8). DPs are loaded in parallel to the task execution, allowing the application to continue processing without waiting for the DPs to finish loading (which is a non-negligible amount of time; it is in the order of milliseconds). This implies that the *motion_estimation* function in the code example (Line 9) will start executing before the DPs have finished loading. Reconfiguring DPs in parallel to task execution provides a speedup for the following reason: if an SI is executed, but not all DPs for the desired implementation are available, the i-Core will use a slower implementation for the same SI that requires only the already loaded DPs (see Section 4.1). When additional DPs are loaded then faster SI implementations become available and they are used automatically.

After completing execution of a particular kernel (e.g. motion_estimation in Line 9), an entirely different set of SIs may be executed on the same share of the fabric attained by the task during its initial *invade* call. The fabric must, however, be prepared for execution of the new SIs (*invade* and *infect*, lines 11-13).

5. Results

In this section, a first evaluation of the i-Core is given, focusing on the application-specific ISE extensions for dif-

ferent tasks. The SIs for speeding up the tasks were developed manually to demonstrate the feasibility and the performance benefits of the i-Core. Figure 7 shows the speedup of four different tasks when accelerated by DPs on the reconfigurable fabric in comparison to executing the tasks without SIs. The obtained speedup depends on the size of the reconfigurable fabric that is expressed by the number of so-called Data Path Containers (DPCs), i.e. regions of the reconfigurable fabric into which a DPs can be reconfigured. For the results in Figure 7, each task uses the available number of DPCs on its own, i.e. the reconfigurable fabric is not shared among multiple tasks. Tasks like the CRC calculation require only few DPs for acceleration. With just one available DPC, a speedup of 2.51x is obtained for CRC. Further DPCs do not lead to further performance benefits for this task. The JPEG decoder approaches its peak performance after 5 DPCs. For more than 5 DPCs (3.49x speedup) only minor additional performance improvements are achieved (up to 3.81x). The image-processing library SUSAN and the H.264 video encoder achieve noticeable performance improvements of more than 18x each. Both tasks consist of multiple kernels that execute repeatedly after each other, i.e. the DPCs are consistently reconfigured to fulfill the task's requirements.

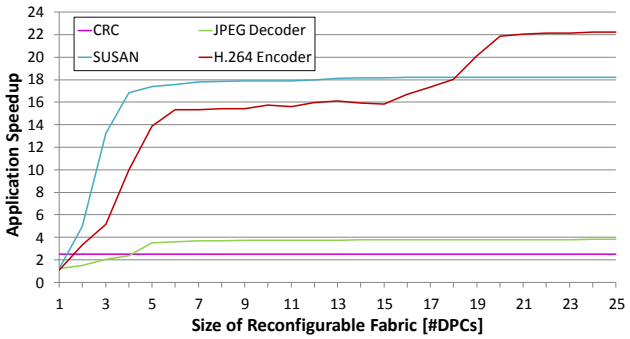

Figure 7: Speedup of different tasks (in comparison to execution without SIs) when executing on the i-Core in single-tasking mode (i.e. the entire reconfigurable fabric is available for the task)

Figure 8 shows the speedup of the same tasks as used in Figure 7, but here all tasks are executed at the same time, i.e. they need to share the available reconfigurable fabric (shown as the horizontal axis). Consequently, the performance improvement for a given number of DPCs is lower than the one shown in Figure 7, as not all available DPCs are assigned to one task. Altogether, five tasks execute, as two instances of the H.264 video encoder are executed at the same time. The figure shows that the characteristics of performance improvement is similar to the case where all tasks execute on their own, i.e. when they can utilize the entire reconfigurable fabric rather than sharing it. This demonstrates that it is possible and beneficial to share the reconfigurable fabric among the tasks.

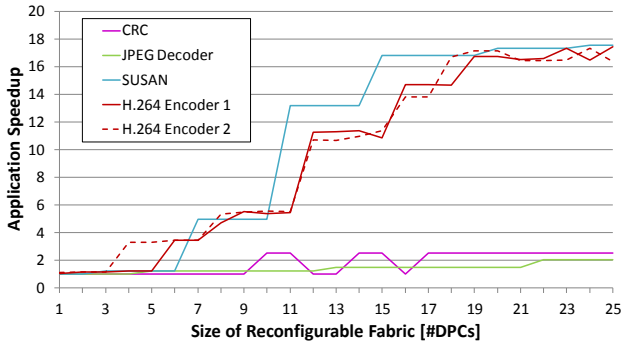

Figure 8: Speedup of different tasks (in comparison to execution without SIs) when executing on the i-Core in multi-tasking mode (i.e. the reconfigurable fabric is shared among all tasks)

6. Conclusion

This paper presented the i-Core concept, a reconfigurable processor that provides a very high adaptivity by utilizing a reconfigurable fabric (to implement Special Instruction) and a reconfigurable microarchitecture. The combination of an adaptive instruction set architecture and microarchitecture allows optimizing performance-wise relevant characteristics of the i-Core to task-specific requirements, which makes the i-Core especially beneficial in multi-tasking scenarios, where different tasks compete for the available resources. We evaluated the i-Core and demonstrated its conceptual advantages when several of these tasks execute together in a multi-tasking environment.

7. Acknowledgement

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89)

References

[1] J. Henkel, "Closing the SoC design gap", *Computer*, vol. 36, no. 9, pp. 119–121, September 2003.

[2] K. Keutzer, S. Malik, and A. R. Newton, "From ASIC to ASIP: The next design discontinuity", in *International Conference on Computer Design (ICCD)*. IEEE Computer Society, September 2002, pp. 84–90.

[3] J. Henkel and S. Parameswaran, *Designing Embedded Processors: A Low Power Perspective*. Springer Publishing Company, Incorporated, 2007.

[4] N. Clark, H. Zhong, and S. Mahlke, "Processor acceleration through automated instruction set customization", in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2003, pp. 129–140.

[5] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints", in *Proceedings of the 40th annual Conference on Design Automation (DAC)*, June 2003, pp. 256–261.

[6] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "A scalable application-specific processor synthesis methodology", in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, November 2003, pp. 283–290.

[7] N. Cheung, J. Henkel, and S. Parameswaran, "Rapid configuration and instruction selection for an ASIP: a case study", in *IEEE/ACM Proceedings of Design Automation and Test in Europe (DATE)*, March 2003, pp. 802–807.

[8] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software", *ACM Computing Surveys (CSUR)*, vol. 34, no. 2, pp. 171–210, June 2002.

[9] F. Barat and R. Lauwereins, "Reconfigurable instruction set processors: A survey", in *Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP)*, June 2000, pp. 168–173.

[10] S. Vassiliadis and D. Soudris, *Fine- and Coarse-Grain Reconfigurable Computing*. Springer Publishing Company, Incorporated, 2007.

[11] H. P. Huynh and T. Mitra, "Runtime adaptive extensible embedded processors – a survey", in *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, July 2009, pp. 215–225.

[12] T. v. Sydow, B. Neumann, H. Blume, and T. G. Noll, "Quantitative analysis of embedded FPGA-architectures for arithmetic", in *Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, September 2006, pp. 125–131.

[13] B. Neumann, T. von Sydow, H. Blume, and T. G. Noll, "Design flow for embedded FPGAs based on a flexible architecture template", in *Proceedings of the conference on Design, automation and test in Europe (DATE)*, 2008, pp. 56–61.

[14] M. Huebner, P. Figuli, R. Girardey *et al.*, "A heterogeneous multicore system on chip with run-time reconfigurable virtual FPGA architecture", in *Proc. of Reconfigurable Architectures Workshop (RAW)*, May 2011.

[15] S. Vassiliadis, S. Wong, G. Gaydadjiev *et al.*, "The MOLEN polymorphic processor", *IEEE Transactions on Computers (TC)*, vol. 53, no. 11, pp. 1363–1375, November 2004.

[16] R. Wittig and P. Chow, "OneChip: an FPGA processor with reconfigurable logic", in *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1996, pp. 126–135.

[17] J. A. Jacob and P. Chow, "Memory interfacing and instruction specification for reconfigurable processors", in *Proceedings of the ACM/SIGDA 7th international symposium on Field Programmable Gate Arrays (FPGA)*, February 1999, pp. 145–154.

[18] J. E. Carrillo and P. Chow, "The effect of reconfigurable units in superscalar processors", in *Proceedings of the international symposium on Field Programmable Gate Arrays (FPGA)*, February 2001, pp. 141–150.

[19] R. Lysecky, G. Stitt, and F. Vahid, "Warp processors", *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 11, no. 3, pp. 659–681, June 2006.

[20] M. Dales, "Managing a reconfigurable processor in a general purpose workstation environment", in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, March 2003, pp. 980–985.

[21] E. Lübbers and M. Platzner, "ReconOS: An RTOS supporting hard- and software threads", in *International Conference on Field Programmable Logic and Applications (FPL)*, August 2007, pp. 441–446.

[22] L. Bauer, M. Shafique, S. Kramer, and J. Henkel, "RISPP: Rotating Instruction Set Processing Platform", in *Proceedings of the 44th annual Conference on Design Automation (DAC)*, June 2007, pp. 791–796.

[23] L. Bauer, M. Shafique, and J. Henkel, "Efficient resource utilization for an extensible processor through dynamic instruction set adaptation", *IEEE Transactions on Very Large Scale Integration Systems (TVLSI), Special Section on Application-Specific Processors*, vol. 16, no. 10, pp. 1295–1308, October 2008.

[24] R. König, L. Bauer, T. Stripf *et al.*, "KAHRISMA: A novel hypermorphic reconfigurable-instruction-set multi-grained-array architecture", in *Proceedings of the 13th conference on Design, Automation and Test in Europe (DATE)*, March 2010, pp. 819–824.

[25] W. Ahmed, M. Shafique, L. Bauer, and J. Henkel, "mRTS: Run-time system for reconfigurable processors with multi-grained instruction-set extensions", in *Proceedings of the 14th conference on Design, Automation and Test in Europe (DATE)*, March 2011, pp. 1554–1559.

[26] SPARC International, Inc., "The SPARC architecture manual, version 8", http://www.sparc.org/specificationsDocuments.html#V8, http://gaisler.com/doc/sparcv8.pdf.

[27] L. Bauer, M. Shafique, and J. Henkel, "A computation- and communication- infrastructure for modular special instructions in a dynamically reconfigurable processor", in *18th International Conference on Field Programmable Logic and Applications (FPL)*, September 2008, pp. 203–208.

[28] J. Teich, J. Henkel, A. Herkersdorf *et al.*, "Invasive computing: An overview", in *Multiprocessor System-on-Chip – Hardware Design and Tool Integration*, M. Hübner and J. Becker, Eds. Springer, Berlin, Heidelberg, 2011, pp. 241–268.

[29] L. Bauer, M. Shafique, and J. Henkel, "Run-time instruction set selection in a transmutable embedded processor", in *Proceedings of the 45th annual Conference on Design Automation (DAC)*, June 2008, pp. 56–61.

[30] R. I. Bahar and S. Manne, "Power and energy reduction via pipeline balancing", in *Proceedings of the International Symp. on Computer architecture (ISCA)*, 2001, pp. 218–229.

[31] S. Ghiasi, J. Casmira, and D. Grunwald, "Using IPC variation in workloads with externally specified rates to reduce power consumption", in *Workshop on Complexity Effective Design*, 2000.

[32] T. Juan, S. Sanjeevan, and J. J. Navarro, "Dynamic history-length fitting: a third level of adaptivity for branch prediction", in *Proceedings of the international symposium on Computer architecture (ISCA)*, 1998, pp. 155–166.

[33] D. H. Albonesi, "Selective cache ways: on-demand cache resource allocation", in *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture (MICRO)*, 1999, pp. 248–259.

[34] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: exploiting generational behavior to reduce cache leakage power", in *Proceedings of the 28th annual international symposium on Computer architecture (ISCA)*, 2001, pp. 240–251.

[35] A. V. Veidenbaum, W. Tang, R. Gupta *et al.*, "Adapting cache line size to application behavior", in *Proceedings of the intl. conference on Supercomputing (ICS)*, 1999, pp. 145–154.

[36] F. Nowak, R. Buchty, and W. Karl, "A run-time reconfigurable cache architecture", in *International Conference on Parallel Computing: Architectures, Algorithms and Applications*, 2007, pp. 757–766.

[37] J. Tao, M. Kunze, F. Nowak *et al.*, "Performance advantage of reconfigurable cache design on multicore processor systems", *International Journal of Parallel Programming*, vol. 36, no. 3, pp. 347–360, 2008.