Probabilistic Alias Analysis for Parallel Programming in SSA Forms

Mohamed A. El-Zawawy¹ and Mohammad N. Alanazi²
¹,²College of Computer and Information Sciences,
Al Imam Mohammad Ibn Saud Islamic University (IMSIU)
Riyadh, Kingdom of Saudi Arabia
¹Department of Mathematics, Faculty of Science
Cairo University
Giza 12613, Egypt
Email¹: maelzawawy@cu.edu.eg
Email²: alanazi@ccis.imamu.edu.sa

Abstract—Static alias analysis of different type of programming languages has been drawing researcher attention. However most of the results of existing techniques for alias analysis are not precise enough compared to needs of modern compilers. Probabilistic versions of these results, in which result elements are associated with occurrence probabilities, are required in optimizations techniques of modern compilers.

This paper presents a new probabilistic approach for alias analysis of parallel programs. The treated parallelism model is that of SPMD where in SPMD, a program is executed using a fixed number of program threads running on distributed machines on different data. The analyzed programs are assumed to be in the static single assignment (SSA) form which is a program representation form facilitating program analysis. The proposed technique has the form of simply-structured system of inference rules. This enables using the system in applications like Proof-Carrying Code (PPC) which is a general technique for proving the safety characteristics of modern programs.

Keywords: Probabilistic Analysis, Alias Analysis, Parallel Programming, SSA Forms.

1. Introduction

Considerable efforts of research have been devoted to achieve the static alias analysis of different type of programming languages. Algorithms for calculating alias relationships for all program points exist for basic programming techniques. Classically, alias relationships fall in two groups: definitely-alias relationships and possibly-alias relationships. The former is typically true for all possible execution paths and the later might typically be true for some of the possible execution paths. However the information calculated by most existing algorithms for alias analysis is not precise enough compared to the needs of modern compilers. This is so as modern compilers need finer alias-information to be able to achieve tasks like code specialization and data speculation. In other words information calculated by most alias analysis techniques do not help compilers to do aggressive optimizations. More specifically, possibly-alias relationships is not rich enough to inform the compiler about the possibility that constraints for the executions. Hence compilers are somehow forced to follow a conservative way and assume the conditions validity for all execution paths [1], [2], [3].

A dominant programming technique of parallelism for large-scale machines equipped with distributed-memories is the single program, multiple data (SPMD) model. In SPMD, a program is executed using a fixed number of program threads running on distributed machines on different data [4]. SPMD can be executed on low-overhead and simple dynamic systems and is convenient for expressing parallelism concepts. This parallelism model is used by message-sending architectures such as MPI. SPMD is also adapted by languages whose address spaces are globally partitioned (PGAS) such as UPC, Co-Array Fortran, and Titanium. Specific deadlocks can be prevented using the SPDM model which can also be used to achieve probabilistic data races and specific program optimizations [5], [6].

Static single assignment (SSA) [7], [8] is a program representation form facilitating program analysis. SSA forms are important for software re-engineering and compiler construction. Program analysis needs data-flow information about points of the program being analyzed. Such information is necessary for program compilation and re-engineering and is conveniently collected by SSA. For program variables, some analyses need to know assignment statements that could have assigned the used variable content. In Static single assignment (SSA) form exactly one variable definition corresponds to a variable use. This is only possible if the algorithm building the SSA form is allowed to insert auxiliary definitions if it is possible for different definitions to get into a specific program point.

A general technique for proving the safety characteristics
of modern programs is Proof-Carrying Code (PCC) [9, 10]. PCC proofs are needed and typically constructed using logics annotated with inference rules that are language-specific. The proofs ensures safety in case there are no bugs in the inference rules. One type of Proof-Carrying Code is Foundational Proof-Carrying Code (FPCC) which uses theories of mathematical logic. The small trusted base of FPCCs and the fact that they are not tied to any specific systems make them more secure and robust.

This paper presents a new technique for probabilistic alias analysis of parallel programs. The technique has the form of simply-strictured system of inference rules. The information calculated by the proposed technique are precise enough compared to information needed by modern compilers for compilations, re-engineering, aggressive-optimization processes. The proposed technique is designed to work on the common and robust data-flow representation; SSA forms of parallel programs. The use of inference systems in the proposed technique makes it straightforward for our technique to produce justifications needed by Foundational Proof-Carrying Code (FPCCs). The proofs have the form of inference rules derivations that are efficiently transferable. The parallelism model treated in this paper is that of single program, multiple data (SPMD) in which the same program is executed on distrusted machines havering different data. However communications between the distributed machines is allowed in a predefined contexts. For example a command running on machine 1 may request machine 3 to evaluate a specific expression using data of machine 3 and to return the result to machine 1.

The syntax of the language used to present the new probabilistic alias analysis technique is shown in Figure 1. We call the language mode SSA-DisLang for ease of reference. A program in SSA-DisLang consists of a sequence of statements where statements are of wide diversity. Statements use (distributed) expressions, DExpr. The machines to run SSA-DisLang programs are typically organized in a hierarchy. The distributed expressions include the following:

- malloc(): allocates a dynamic array in memory and return its base address.
- run (e, m): evaluates the distributed expression e on machine m and return the result.
- reform(alis m, int m) e: cast the location denoted by the distributed expression e as an integer rather than a pointer to a memory location on machine m.
- reform (int m, int m) e: cast the location denoted by the distributed expression e as a pointer a memory location on machine m rather than as a pointer to a memory location on machine m.

Our proposed technique assumes that the given program, that is to be be analyzed for its probabilistic alias competent, has the static single assignments form. Therefore the input program would contain annotations (added by any efficient SSA such as algorithm [11]). The program annotations will have the form of new statements added to the original program. Therefore statements of SSA-DisLang include the following:

- l := e: this is a classical assignment command. However the design of the language allows using this command to assign a value evaluated at a machine to a location on a different machine of the machines hierarchy.
- run (S, m): allows evaluates a specific command S on a specific machine m regardless of the executing machine. This command is necessary when some commands are convenient only to run on data of certain machine of the hierarchy. The command is also used when security is a concern as S would not have access to all machines.
- \( x_i := f(x_j, x_k) \): this command is to be added by the supposed SSA algorithm and it semantics is that variable \( x_i \) were created specifically for avoiding multiple assignments to variable \( x \). The range of this definition if form definition of variable \( x_j \) to that of variable \( x_k \).
- \( x_i := md(x_j) \): this is the second sort of annotations SSA-DisLang programs. The semantics of this statement is that it is highly likely that variable \( x_i \) is used to define variable \( x_j \). Recall that our main technique of the paper cares about possibility of assignments to occur in

Motivation

The paper is motivated by need for a precise probabilistic alias analysis for SPMD programs running on a hierarchy of distributed machines. The required technique is supposed to associate each analysis result with a correctness proof (in the form of type derivations) to be used in proof-carrying code applications.

Contributions

The contribution of the paper is a new approach for probabilistic alias analysis of SPMD programs running on SSA forms of programs and producing justifications with analysis results.

Paper Outline

The outline of this paper is as follows. Section 2 presents the language model, SSA-DisLang, of the paper. This section also presents an informal semantics to the language constructs. The main content of Section 2 is the new technique of the probabilistic alias analysis of SPMD programs. Section 4 concludes the paper and suggests directions for future work.

2. Probabilistic Alias analysis for SPMD

This section presents a new technique for probabilistic alias analysis of parallel programs. The parallelism model used here is that of SPMD where the same program is executed on distrusted machines havering different data.
\[ I \in \text{Loc} \quad ::= \quad x \mid l \to y \mid [l]. \]
\[ e \in \text{DExp} \quad ::= \quad l \mid e_1 \text{ iop } e_2 \mid \& l \mid \text{malloc()} \mid \text{run} \ (e, m) \mid \text{reform} (\text{alis } m, \text{int } m) \ e \mid \text{reform} (\text{int } m_j, \text{int } m_i) \ e. \]
\[ S \in \text{Stmts} \quad ::= \quad l := e \mid \text{run} \ (S, m) \mid S_1 ; S_2 \mid x_i := f(x_j, x_k) \mid x_i := \text{md}(x_j) \mid \text{mu}(x_j) \mid \text{if } e \ \text{then } S_t \ \text{else } S_f \mid \text{while } e \ \text{do } S_t. \]

Fig. 1: Programming Language Model; SSA-DisLang

\[
P(x) = \{(a_1, p_1), \ldots, (a_n, p_n)\} \quad i = \max(p_1, \ldots, p_n) \quad \frac{x : P \rightarrow_l a_i}{(x^p)}
\]

\[
P(l) = \{(a_1, p_1), \ldots, (a_n, p_n)\} \quad i = \max(p_1 \times q_1 \mid a_j \in P(y)_{1\ldots i}) \quad P(y) = \{(b_1, q_1), \ldots, (b_m, q_m)\} \quad \frac{(l \rightarrow y) : P \rightarrow_l b_i}{(\rightarrow^p)}
\]

\[
P(l) = \{(a_1, p_1), \ldots, (a_n, p_n)\} \quad \forall a_i, P(a_i) = \{(b_1^i, q_1^i), \ldots, (b_m^i, q_m^i)\} \quad \frac{i = \max(p_1 \times q_1 \mid 1 \leq i \leq n \& 1 \leq j \leq m)}{[l] : P \rightarrow_l b_i}
\]

Fig. 2: Probabilistic Alias Analysis (PAA): Locations.

\[
e : P \rightarrow a_e \quad \text{probability of arriving at this memory point} \geq p_t \quad \frac{\text{reform} (\text{alis } m \rightarrow \text{int } m) \ e : P \rightarrow_l a_e}{(\text{reform}^p \ e)}
\]

\[
e : P \rightarrow a_e \quad \text{probability of arriving at this memory point} < p_t \quad \frac{\text{reform} (\text{alis } m \rightarrow \text{int } m) \ e : P \rightarrow_l \perp}{(\text{reform}^p \ e)}
\]

\[
e : P \rightarrow a_e \quad \text{probability of arriving at this memory point} \geq p_t \quad \frac{\text{reform} (\text{int } m_j \rightarrow \text{int } m_i) \ e : P \rightarrow_l a_e}{(\text{reform}^p \ e)}
\]

\[
e : P \rightarrow a_e \quad \text{probability of arriving at this memory point} < p_t \quad \frac{\text{reform} (\text{int } m_j \rightarrow \text{int } m_i) \ e : P \rightarrow_l \perp}{(\text{reform}^p \ e)}
\]

\[
e : P \rightarrow a_e \quad b = \text{reform} (\_, \text{int } m) a_e \quad \frac{\text{run} \ (e, m) : P \rightarrow_l b}{(\text{run}^p \ b)}
\]

\[
a_e \text{ is a fresh memory location on machine } m_i \quad \frac{\text{malloc()} : P \rightarrow_l a_i}{(\text{malloc}^p)}
\]

\[
e_1 : P \rightarrow a_{e_1} \quad e_2 : P \rightarrow a_{e_2} \quad \frac{e_1 i_{op} e_2 : P \rightarrow_l a_{e_1} + a_{e_2}}{(\_^p)}
\]

Fig. 3: Probabilistic Alias Analysis (PAA): Distributed Expressions.
The type operational semantics for is captured as and an alias type state of the type empty domain is denoted by of the language semantics. Simplicity of the theorem proof deeply relies on the choice authors’s experience and based on some experiments, the only reference to the semantics in this paper. From the Many semantics candidates exist. Due to lack of space we SSA-DisLang the following theorem. The theorem requests the existence of SSA-DisLang figure 3. Some comments on the rules are in order. The rules with the required probabilistic alias information. A probabilistic pair is a variable (register) or a memory location and a number p such that 0 ≤ p ≤ 1. Decision produced by the system have the forms e : P → a and S : P → P’. The judgement e : P → a means that evaluating the expression e in a memory state of the type P results in the memory address a. The semantics of the judgement e : P → a is that running S in a memory state of the type P results (if ends) in a memory state of the type P’. The proposed technique is meant to be used as follows. Given a distributed program S, one constructs (using inference rules of the system) an alias type P’ such that S : ⊥ → P’. The base type is the partial map with an empty domain is denoted by ⊥. The construction of P’ is a type derivation process and results in annotating program with the required probabilistic alias information.

Inference rules for distributed expressions are shown in figure 4. Some comments on the rules are in order. The rules for expressions only considers the address evaluated from e if there is a considerable probability (probability threshold > p_m) of arriving at the concerned program point. Inference rules for statements are shown in figure 4. Some comments on the rules are in order. The rule (\(\frac{\text{e}}{\text{a}}\)) uses the base address of the denoted by l and the address returned for e by the inference rules of expressions. The image of these addresses under the pre-type also contribute to calculating the post type of the de-reference statement.

The soundness of our proposed technique is guaranteed by the following theorem. The theorem requests the existence of robust operational semantics for the language SSA-DisLang. Many semantics candidates exist. Due to lack of space we only reference to the semantics in this paper. From the authors’s experience and based on some experiments, the simplicity of the theorem proof deeply relies on the choice of the language semantics.

**Theorem 1:** Suppose that S is a SSA-DisLang program and S : ⊥ → P’. Suppose also that using a convenient operational semantics for SSA-DisLang, the execution of S is captured as S : M → M’. Then the final memory state M’ is of the the probabilistic alias type P’.

### 3. Related Work

The changing associations characteristics property of pointers makes the points-to analysis a complicated problem [3]. Much research [12], [13], [14], [15] have been developed to solve the pointer analysis problem. Each of these techniques evaluates either points-to or aliases relationships at program points. Points-to and aliases relationships are classified into two classes: definitely-aliases (or must-points-to) relationships and may-points-to (or possibly-aliases relationships). While the later relationships are true on some executions, the former relationships are true on all executions. Whether possibly-aliases or may-points-to relationships are true on most executions or on few executions is not measurable by most of these techniques. For specific transformations and optimizations these missed information are beneficial. Few attempts were made to fill this gap.

Using traditional data-flow analysis, in [16], [17] a theoretical formulation is presented to compute measurable information. More specifically, this work evaluates, for each program point, the predicted count that specific conditions may hold. Aiming at evaluating, among array references, the probabilities of aliases, [18], [19], [20] presents a probabilistic technique for memory disambiguation. A probabilistic, interprocedural, context-sensitive, and flow-sensitive techniques for alias analysis were proposed in [3], [11], [21]. On alias relationships, these technique evaluate measurable information. MachSUIF and SUIF compiler infrastructures provided the bases for the implementation of these techniques. The probabilities of pointer induced, loop carried, and data dependence relationships were evaluated in [22], [23]. Using sparse matrices, as efficient linear transfer functions, [24], [25] modeled probabilistic alias analysis. The results of this research were proved accurate. [26] presents an algorithm to evaluate measurable alias information. A technique for memory disambiguation, evolution of probabilities that pairs of memory pointers point at the same memory location, is presented in [26].

For array optimizations and analysis, probabilistic techniques for memory disambiguation were proposed [18]. These techniques typically present data speculations [27] necessary for modern architectures of computers. For distributed parallel machines with shared-memory, an important problem is that of compiler optimizations for programs that are pointer-based. This is so as the host processor of an object can be determined using data distribution analysis [28] and affinity analysis [29].

In, pointer-based programs, a reference is referencing a group of objects with may-points-to. For such cases, traditional affinity analysis [30] can be integrated with traditional pointer analysis. The result of this integration is a technique that evaluates the parts of objects on a processor’s list.
\[ P(x_j) = (x_i, a) \quad P(x_1) = (x_k, \ldots) \]
\[ x_i := \text{md}(f(x_i) : P \rightarrow_s P[x_i \mapsto \{(x_k, a), (x_j, 1 - a)\}]) \]
\[ (\text{md}^p) \]

\[ p_j(p_k) \text{ is the probability of executing the path from definition of } x_j(x_k) \text{ to that of } x_i \]
\[ x_i := \text{fi}(x_j(x_k) : P \rightarrow_s P[x_i \mapsto \{(x_j, p_j), (x_k, p_k)\}]) \]
\[ (fi^p) \]

\[ a \text{ is the base address of array } l \]
\[ x := \&l : P \rightarrow_s P[x \mapsto \{(a, 1)\}] \]
\[ (\&_1) \]
\[ \mu(x_j) : P \rightarrow_s P \]
\[ (\mu^p) \]

\[ a_1, a_2 \text{ are the base addresses of arrays } l_1 \text{ and } l_2 \]
\[ i \text{ is the index of } y \text{ in } l_2 \]
\[ l_1 \rightarrow y := \&l_2 : P \rightarrow_s P[a_1 \mapsto \{(a_2 + i, 1)\}] \]
\[ (\&_2) \]

\[ a_1, a_2 \text{ are the base addresses of arrays } l_1 \text{ and } l_2 \]
\[ P(a_1) = \{(b_1, p_1), \ldots, (b_n, p_n)\} \]
\[ (\&^p) \]

\[ l \neq [\ldots] \]
\[ a_l \text{ is the base addresses of array } l \]
\[ e : P \rightarrow a_e \]
\[ l := [e] : P \rightarrow_s P[a_l \mapsto \{(b_1, p_1), \ldots, (b_n, p_n)\}] \]
\[ (\text{(P1)}^p) \]

\[ e \neq [\ldots] \]
\[ a_l \text{ is the base addresses of array } l \]
\[ e : P \rightarrow a_e \]
\[ P(a_e) = \{(b_1, p_1), \ldots, (b_n, p_n)\} \]
\[ P(a_l) = \{(c_1, q_1), \ldots, (c_m, q_m)\} \]
\[ \forall c, p(b_i) = \{(d_i^l, t_i^l), \ldots, (q_i^l, t_i^l)\} \]
\[ \text{if } c, p(b_i) \text{ then } e \text{ else } a_e \]
\[ (\text{(P2)}^p) \]

\[ a_l \text{ is the base addresses of array } l \]
\[ e : P \rightarrow a_e \]
\[ l := [e] : P \rightarrow_s P[c_l \mapsto \{(b_1, \text{min}(p_1, q_1)), \ldots, (b_n, \text{min}(p_n, q_n))\} \mid 1 \leq i \leq m] \]
\[ (\text{(P3)}^p) \]

\[ e \neq [\ldots] \]
\[ l \neq [\ldots] \]
\[ e \neq \&e \ldots \]
\[ a_l \text{ is the base addresses of array } l \]
\[ l := e : P \rightarrow_s P[a_l \mapsto \{(b_1, p_1), \ldots, (b_n, p_n)\}] \]
\[ (:=^p) \]

\[ S_1 : P \rightarrow_{s} P' \]
\[ S_2 : P'' \rightarrow_{s} P' \]
\[ S_1; S_2 : P \rightarrow_{s} P' \]
\[ (p) \]
\[ S : P \rightarrow_{s} P' \]
\[ \text{run} \ (S, m) : P \rightarrow_{s} P' \]
\[ (\text{run}^p) \]
\[ S_1 : P \rightarrow_{s} P_t \]
\[ S_f : P_f \rightarrow_{s} P' \]
\[ \text{if } e \text{ then } S_t \text{ else } S_f : P \rightarrow_{s} P_t \]
\[ (\text{if}^p) \]
\[ n \text{ is the expected execution time of } S_t \]
\[ n \leq S_t : P \rightarrow_{s} P_t \]
\[ (\text{whl}^p) \]

Fig. 4: Probabilistic Alias Analysis (PAA): Statements.
of task executions. This is necessary for many program optimizations.

There are many examples of aggressive optimizations such as data speculations, speculative multithreading (thread partitioning), and code specialization [31], [32]. To boost performance of modern architectures, these optimizations are typically achieved by compilers. Compilers can only do such tasks if they are able to measure the possibility of dynamic pointer associations. Using interval analysis, irreducible flow graphs, and the elimination technique, intraprocedural analysis can be used to handle pointer analysis of programs [33].

Extensions to such techniques to cover context-sensitive analysis that is interprocedural is achievable as well.

Examples of analysis for speculative multithreading model include thread partitioning [34], [35], [36]. Such analysis boosts compilers performance via running speculative threads in case of low possibilities for conflicts. In this scenario for threads with high possibilities are turned off [22].

4. Conclusion and Future Work

This paper presented a new technique for probabilistic alias analysis of SPMD programs. The new approach has the form of system of inference rules. This has direct applications in proof-carrying code area of research. The proposed technique also has the advantage of assuming SSA forms of analyzed programs.

Directions for future work include the following. Producing probabilistic techniques for important analyses (such as dead-code elimination) for SPMD programs that uses the results of the analysis proposed in this paper would be an important contribution. Producing other analyses for the language model of this paper in the spirit of [37], [38], [39] is another direction for future work. There is also a need for precise probabilistic operational semantics for SPMD programs. This semantics would be important to accurately measure probabilities of statements executions and probabilities of executions order.

5. Acknowledgment

The authors acknowledge the support (grants numbers 340918 & 330911) of the deanship of scientific research of Al Imam Mohammad Ibn Saud Islamic University (IMSIU).

References


