Probabilistic Alias Analysis for Parallel Programming in SSA Forms

Mohamed A. El-Zawawy[1] **and Mohammad N. Alanazi**[2]
[1,2]College of Computer and Information Sciences,
Al Imam Mohammad Ibn Saud Islamic University (IMSIU)
Riyadh, Kingdom of Saudi Arabia
[1]Department of Mathematics, Faculty of Science
Cairo University
Giza 12613, Egypt
Email[1]: maelzawawy@cu.edu.eg
Email[2]: 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-strictured 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 comfier 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 different machines on different sets of data.

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 langauge model, *SSA-DisLang*, of the paper. This section also presents an informal semantics to the langauge 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. 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 langauge used to present the new probabilistic alias analysis technique is shown in Figure 1. We call the langauge 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 : casts 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_j, int m_i)) e : casts the location denoted by the distributed expression e as a pointer a memory location on machine m_i rather than as a pointer to a memory location on machine m_j.

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 langauge 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

$$
\begin{array}{rcl}
 & & x \in \text{IVar},\ i_{op} \in I_{op},\ b_{op} \in B_{op},\ \text{and } m \in M \subseteq \mathcal{M} \\
l \in \text{Loc} & ::= & x \mid l \to y \mid [l]. \\
e \in \text{DExpr} & ::= & l \mid e_1\ i_{op}\ e_2 \mid \&l \mid \text{malloc}() \mid \text{run } (e, m) \mid \\
 & & \text{reform(alis } m, \text{int } m)\ e \mid \text{reform (int } m_j, \text{int } m_i))\ e. \\
S \in \text{Stmts} & ::= & l := e \mid \text{run } (S, m) \mid S_1; S_2 \mid x_i := f(x_j, x_k) \mid x_i := md(x_j) \mid \\
 & & mu(x_j) \mid \text{if } e \text{ then } S_t \text{ else } S_f \mid \text{while } e \text{ do } S_t.
\end{array}
$$

Fig. 1: Programming Language Model; *SSA-DisLang*

$$
\frac{P(x) = \{(a_1, p_1), \ldots, (a_n, p_n)\} \qquad i = \max(p_1, \ldots, p_n)}{x : P \to_l a_i}\ (\text{x}^p)
$$

$$
\frac{\begin{array}{l} P(l) = \{(a_1, p_1), \ldots, (a_n, p_n)\} \qquad P(y) = \{(b_1, q_1), \ldots, (b_m, q_m)\} \\ i = \max\{p_j \times q_j \mid a_j \in P(y)\rceil_1\} \end{array}}{(l \to y) : P \to_l b_i}\ (\to^p)
$$

$$
\frac{\begin{array}{l} P(l) = \{(a_1, p_1), \ldots, (a_n, p_n)\} \qquad forall i.P(a_i) = \{(b^i_1, q^i_1), \ldots, (b^i_m, q^i_m)\} \\ i = \max\{p_i \times q^i_j \mid 1 \leq i \leq n \& 1 \leq j \leq m\} \end{array}}{[l] : P \to_l b_i}\ ([l]^p)
$$

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

$$
\frac{e : P \to a_e \qquad \text{probability of arriving at this memory point} \geq p_t}{\text{reform(alis } m \to \text{int } m)\ e : P \to_l a_e}\ (\text{reform}^p_1)
$$

$$
\frac{\text{probability of arriving at this memory point} < p_t}{\text{reform(alis } m \to \text{int } m)\ e : P \to_l \perp}\ (\text{reform}^p_2)
$$

$$
\frac{e : P \to a_e \qquad \text{probability of arriving at this memory point} \geq p_t}{\text{reform (int } m_j \to \text{int } m_i))\ e : P \to_l a_e}\ (\text{reform}^p_3)
$$

$$
\frac{\text{probability of arriving at this memory point} < p_t}{\text{reform (int } m_j \to \text{int } m_i))\ e : P \to_l \perp}\ (\text{reform}^p_4)
$$

$$
\frac{e : P \to a_e \qquad b = \text{reform}(\_, \text{int } m) a_e}{\text{run } (e, m) : P \to_l b}\ (\text{run}^p_e)
$$

$$
\frac{a_i \text{ is a fresh memory location on machine } m_i}{\text{malloc}() : P \to_l a_i}\ (\text{malloc})^p
$$

$$
\frac{e_1 : P \to a_{e_1} \qquad e_2 : P \to a_{e_2}}{e_1\ i_{op}\ e_2 : P \to_l a_{e_1} + a_{e_2}}\ (+^p)
$$

Fig. 3: Probabilistic Alias Analysis (PAA): Distributed Expressions.

percentages; it is not a $0/1$ technique.

- $mu(x_j)$: this is the third and last sort of annotations *SSA-DisLang* programs. The semantics of this command is that variable x_j is highly likely to be used in the following de-reference command of the program.

Figures 2, 3, and 4 present elements of our proposed technique for probabilistic alias analysis of *SSA-DisLang* programs. The proposed technique has the form of a type system which consists of set of alias types denoted by P and set of inference rules presented in the technique figures. An alias type is a *partial* map. The domain of this partial map is a subset of the set of all variables (denoting registers) allowed to be used on different machines of the distributed hierarchy plus the set of all addresses of memories of machines on hierarchy. The codomain of the alias type is the power set of the set of all *probabilistic pairs*. A probabilistic pair is a pair of variable (register) or a memory location and a number p such hat $0 \leq p \leq 1$.

Judgment produced by the system have the forms $e : P \rightarrow a$ and $S : P \rightarrow P'$. The judgement $e : P \rightarrow a$ 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 $e : P \rightarrow a$ 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 $S : \perp \rightarrow P'$. The base type is the partial map with an empty domain is denoted by $\perp$. 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 3. Some comments on the rules are in order. The rules for *reform* expressions only considers the address evaluated from e if there is a considerable probability (probability threshold $> p_{th}$) 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 $([]_3^p)$ uses the base address of the array 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 guarantied by the following theorem. The theorem requests the existence of robust operational semantics for the langauge *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 langauge semantics.

Theorem 1: Suppose that S is a *SSA-DisLang* program and $S : \perp \rightarrow P'$. Suppose also that using a convenient operational semantics for *SSA-DisLang*, the execution of S is captured as $S : M \rightarrow M'$. 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. Wether 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, contextsensitive, 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

$$\frac{P(x_j) = (x_i, a) \qquad P(x_i) = (x_k, \_)}{x_i := md(x_j) : P \to_s P[x_i \mapsto \{(x_k, a), (x_j, 1-a)\}]} \; (md^p)$$

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

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

$$\frac{\begin{array}{l} a_1, a_2 \text{ are the base addresses of arrays } l_1 \text{ and } l_2 \\ i \text{ is the index of } y \text{ in } l_2 \end{array}}{l_1 \to y := \&l_2 : P \to_s P[a_1 \mapsto \{(a_2 + i, 1)\}]} \; (\&_2^p)$$

$$\frac{\begin{array}{l} a_1, a_2 \text{ are the base addresses of arrays } l_1 \text{ and } l_2 \\ P(a_1) = \{(b_1, p_1), \ldots, (b_n, p_n)\} \end{array}}{[l_1] := \&l_2 : P \to_s P[b_1 \mapsto \{(a_2, p_1)\}, \ldots, b_n \mapsto \{(a_2, p_n)\}]} \; (\&_3^p)$$

$$\frac{\begin{array}{ll} l \neq [\ldots] & e : P \to a_e \\ a_l \text{ is the base addresses of array } l & P(a_e) = \{(b_1, p_1), \ldots, (b_n, p_n)\} \end{array}}{l := [e] : P \to_s P[a_l \mapsto \{(b_1, p_1), \ldots, (b_n, p_n)\}]} \; ([]_1^p)$$

$$\frac{\begin{array}{ll} e \neq [\ldots] & \\ a_l \text{ is the base addresses of array } l & P(a_e) = \{(b_1, p_1), \ldots, (b_n, p_n)\} \\ e : P \to a_e & P(a_l) = \{(c_1, q_1), \ldots, (c_m, q_m)\} \end{array}}{[l] := e : P \to_s P[c_i \mapsto \{(b_1, \min(p_1, q_1)), \ldots, (b_n, \min(p_n, q_n))\} \mid 1 \leq i \leq m]} \; ([]_2^p)$$

$$\frac{\begin{array}{ll} & P(a_e) = \{(b_1, p_1), \ldots, (b_n, p_n)\} \\ a_l \text{ is the base addresses of array } l & \forall i. p(b_i) = \{(d_1^i, t_1^i), \ldots, (q_k^i, t_k^i)\} \\ e : P \to a_e & P(a_l) = \{(c_1, q_1), \ldots, (c_m, q_m)\} \end{array}}{[l] := [e] : P \to_s P[c_i \mapsto \{(d_1^i, \min(p_1, q_1 \times t_1^i)), \ldots, (q_k^i, \min(p_n, q_n \times t_k^i))\} \mid 1 \leq i \leq m]} \; ([]_3^p)$$

$$\frac{\begin{array}{ll} e \neq [\ldots] \quad l \neq [\ldots] \quad e \neq \& \ldots & e : P \to a_e \\ a_l \text{ is the base addresses of array } l & P(a_e) = \{(b_1, p_1), \ldots, (b_n, p_n)\} \end{array}}{l := e : P \to_s P[a_l \mapsto \{(b_1, p_1), \ldots, (b_n, p_n)\}]} \; (:=^p)$$

$$\frac{\begin{array}{c} S_1 : P \to_s P'' \\ S_2 : P'' \to_s P' \end{array}}{S_1; S_2 : P \to_s P'} \; (;^p) \qquad \frac{S : P \to_s P'}{\text{run } (S, m) : P \to_s P'} \; (\text{run}_s^p)$$

$$\frac{S_t : P \to_s P_t \qquad S_f : P_f \to_s P'}{\text{if } e \text{ then } S_t \text{ else } S_f : P \to_s P_t \biguplus P_f} \; (\text{if}^p)$$

$$\frac{n \text{ is the exepcted execution time of } S_t \qquad S_t : P \to_s P_t}{\text{while } e \text{ do } S_t : P \to_s \biguplus_n P_t} \; (\text{whl}^p)$$

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 langauge 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

[1] U. P. Khedker, A. Mycroft, and P. S. Rawat, "Liveness-based pointer analysis," in *SAS*, ser. Lecture Notes in Computer Science, A. Miné and D. Schmidt, Eds., vol. 7460. Springer, 2012, pp. 265–282.

[2] M. A. El-Zawawy, "Flow sensitive-insensitive pointer analysis based memory safety for multithreaded programs," in *ICCSA (5)*, ser. Lecture Notes in Computer Science, B. Murgante, O. Gervasi, A. Iglesias, D. Taniar, and B. O. Apduhan, Eds., vol. 6786. Springer, 2011, pp. 355–369.

[3] P.-S. Chen, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee, "Interprocedural probabilistic pointer analysis," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 10, pp. 893–907, 2004.

[4] P. Pacheco, *An Introduction to Parallel Programming*. Elsevier, 2011, 1 edition (2011).

[5] H. Li, G. Fox, G. von Laszewski, and A. Chauhan, "Co-processing spmd computation on cpus and gpus cluster," in *CLUSTER*. IEEE, 2013, pp. 1–10.

[6] M. Tsuji, M. Sato, M. R. Hugues, and S. G. Petiton, "Multiple-spmd programming environment based on pgas and workflow toward post-petascale computing," in *ICPP*. IEEE, 2013, pp. 480–485.

[7] W. Amme, T. S. Heinze, and J. von Ronne, "Intermediate representations of mobile code," *Informatica (Slovenia)*, vol. 32, no. 1, pp. 1–25, 2008.

[8] W. Amme, N. Dalton, M. Franz, and J. von Ronne, "Safetsa: A type safe and referentially secure mobile-code representation based on static single assignment form," in *PLDI*, M. Burke and M. L. Soffa, Eds. ACM, 2001, pp. 137–147.

[9] F. Pfenning, L. Caires, and B. Toninho, "Proof-carrying code in a session-typed process calculus," in *CPP*, ser. Lecture Notes in Computer Science, J.-P. Jouannaud and Z. Shao, Eds., vol. 7086. Springer, 2011, pp. 21–36.

[10] R. Jobredeaux, H. Herencia-Zapana, N. A. Neogi, and E. Feron, "Developing proof carrying code to formally assure termination in fault tolerant distributed controls systems," in *CDC*. IEEE, 2012, pp. 1816–1821.

[11] M.-Y. Hung, P.-S. Chen, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee, "Support of probabilistic pointer analysis in the ssa form," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 12, pp. 2366–2379, 2012.

[12] Y. Ben-Asher and N. Rotem, "Using memory profile analysis for automatic synthesis of pointers code," *ACM Trans. Embedded Comput. Syst.*, vol. 12, no. 3, p. 68, 2013.

[13] S. Staiger-Stöhr, "Practical integrated analysis of pointers, dataflow and control flow," *ACM Trans. Program. Lang. Syst.*, vol. 35, no. 1, p. 5, 2013.

[14] L. Li, C. Cifuentes, and N. Keynes, "Precise and scalable context-sensitive pointer analysis via value flow graph," in *ISMM*, P. Cheng and E. Petrank, Eds. ACM, 2013, pp. 85–96.

[15] B. Huang, X. Ling, and G. Wu, "Field-sensitive function pointer analysis using field propagation for state graph extraction," *JSW*, vol. 8, no. 7, pp. 1592–1603, 2013.

[16] Q. Shao, Y. D. Chen, and L. Zhang, "An extension of three-parameter burr iii distribution for low-flow frequency analysis," *Computational Statistics & Data Analysis*, vol. 52, no. 3, pp. 1304–1314, 2008.

[17] F. Miwakeichi, P. A. Valdes-Sosa, E. Aubert-Vazquez, J. B. Bayard, J. Watanabe, H. Mizuhara, and Y. Yamaguchi, "Decomposing eeg data into space-time-frequency components using parallel factor analysis and its relation with cerebral blood flow," in *ICONIP (1)*, ser. Lecture Notes in Computer Science, M. Ishikawa, K. Doya, H. Miyamoto, and T. Yamakawa, Eds., vol. 4984. Springer, 2007, pp. 802–810.

[18] R. D.-C. Ju, J.-F. Collard, and K. Oukbir, "Probabilistic memory disambiguation and its application to data speculation," *SIGARCH Computer Architecture News*, vol. 27, no. 1, pp. 27–30, 1999.

[19] B. Guo, Y. Wu, C. Wang, M. J. Bridges, G. Ottoni, N. Vachharajani, J. Chang, and D. I. August, "Selective runtime memory disambiguation in a dynamic binary translator," in *CC*, ser. Lecture Notes in Computer Science, A. Mycroft and A. Zeller, Eds., vol. 3923. Springer, 2006, pp. 65–79.

[20] C. Fang, S. Carr, S. Önder, and Z. Wang, "Feedback-directed memory disambiguation through store distance analysis," in *ICS*, G. K. Egan and Y. Muraoka, Eds. ACM, 2006, pp. 278–287.

[21] A. D. Pierro, C. Hankin, and H. Wiklicky, "A systematic approach to probabilistic pointer analysis," in *APLAS*, ser. Lecture Notes in Computer Science, Z. Shao, Ed., vol. 4807. Springer, 2007, pp. 335–350.

[22] P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee, "Compiler support for speculative multithreading architecture with probabilistic points-to analysis," in *PPOPP*, R. Eigenmann and M. C. Rinard, Eds. ACM, 2003, pp. 25–36.

[23] A. Zhai, J. G. Steffan, C. B. Colohan, and T. C. Mowry, "Compiler and hardware support for reducing the synchronization of speculative threads," *TACO*, vol. 5, no. 1, 2008.

[24] J. D. Silva and J. G. Steffan, "A probabilistic pointer analysis for

speculative optimizations," in *ASPLOS*, J. P. Shen and M. Martonosi, Eds. ACM, 2006, pp. 416–425.

[25] S. Roy and Y. N. Srikant, "The hot path ssa form: Extending the static single assignment form for speculative optimizations," in *CC*, ser. Lecture Notes in Computer Science, R. Gupta, Ed., vol. 6011. Springer, 2010, pp. 304–323.

[26] Y.-M. Lu and P.-S. Chen, "Probabilistic alias analysis of executable code," *International Journal of Parallel Programming*, vol. 39, no. 6, pp. 663–693, 2011.

[27] L. Xiang and M. L. Scott, "Compiler aided manual speculation for high performance concurrent data structures," in *PPOPP*, A. Nicolau, X. Shen, S. P. Amarasinghe, and R. Vuduc, Eds. ACM, 2013, pp. 47–56.

[28] J. K. Lee, D. Ho, and Y. C. Chuang, "Data distribution analysis and optimization for pointer-based distributed programs," in *ICPP*. IEEE Computer Society, 1997, pp. 56–63.

[29] M. C. Carlisle and A. Rogers, "Software caching and computation migration in olden," *J. Parallel Distrib. Comput.*, vol. 38, no. 2, pp. 248–255, 1996.

[30] T. Pitkäranta, "Affinity analysis of coded data sets," in *EDBT/ICDT Workshops*, ser. ACM International Conference Proceeding Series, M. Mesiti, T. M. Truta, L. Xiong, S. Müller, H. Naacke, B. Novikov, G. Raschia, I. Sanz, P. Sens, D. Shaporenkov, and N. Travers, Eds., vol. 360. ACM, 2009, pp. 177–184.

[31] R. Zhang, S. Debray, and R. T. Snodgrass, "Micro-specialization: dynamic code specialization of database management systems," in *CGO*, C. Eidt, A. M. Holler, U. Srinivasan, and S. P. Amarasinghe, Eds. ACM, 2012, pp. 63–73.

[32] M. A. Khan, "Feedback-directed specialization of code," *Computer Languages, Systems & Structures*, vol. 36, no. 1, pp. 2–15, 2010.

[33] Q. Sun, J. Zhao, and Y. Chen, "Probabilistic points-to analysis for java," in *CC*, ser. Lecture Notes in Computer Science, J. Knoop, Ed., vol. 6601. Springer, 2011, pp. 62–81.

[34] B. Liu, Y. Zhao, Y. Li, Y. Sun, and B. Feng, "A thread partitioning approach for speculative multithreading," *The Journal of Supercomputing*, vol. 67, no. 3, pp. 778–805, 2014.

[35] Y. Li, Y. Zhao, P. Yin, and Y. Du, "Speculative thread partitioning using fuzzy c-means clustering," in *CSE*, W. Qu, K. Lin, Y. Shen, W. Shi, D. F. Hsu, X. Jin, F. C. M. Lau, and J. Xu, Eds. IEEE, 2011, pp. 199–206.

[36] W.-J. Kim, K. Cho, and K.-S. Chung, "Multi-threaded syntax element partitioning for parallel entropy decoding," *IEEE Trans. Consumer Electronics*, vol. 57, no. 2, pp. 897–905, 2011.

[37] M. A. El-Zawawy and H. A. Nayel, "Type systems based data race detector," *IJCSNS International Journal of Computer Science and Network Security*, vol. 5, no. 4, pp. 53–60, July 2012.

[38] M. A. El-Zawawy and N. M. Daoud, "New error-recovery techniques for faulty-calls of functions," *Computer and Information Science*, vol. 5, no. 3, pp. 67–75, May 2012.

[39] M. A. El-Zawawy and H. A. Nayel, "Partial redundancy elimination for multi-threaded programs," *IJCSNS International Journal of Computer Science and Network Security*, vol. 11, no. 10, pp. 127–133, October 2011.