RESEARCH ARTICLE

ScienceAsia 37 (2011): **344–354**

doi: 10.2306/scienceasia1513-1874.2011.37.344

Probabilistic pointer analysis for multithreaded programs

Mohamed A. El-Zawawy

College of Computer and Information Sciences, Al-Imam M.I.-S.I. University, Riyadh 11432, Kingdom of Saudi Arabia
Department of Mathematics, Faculty of Science, Cairo University, Giza 12613, Egypt

e-mail: maelzawawy@cu.edu.eg

Received 13 Apr 2011
Accepted 3 Nov 2011

ABSTRACT: The use of pointers and data-structures based on pointers results in circular memory references that are interpreted by a vital compiler analysis, namely pointer analysis. For a pair of memory references at a program point, a typical pointer analysis specifies if the points-to relation between them may exist, definitely does not exist, or definitely exists. The "may be" case, which describes the points-to relation for most of the pairs, cannot be dealt with by most compiler optimizations. This is so to guarantee the soundness of these optimizations. However, the "may be" case can be capitalized by the modern class of speculative optimizations if the probability that two memory references alias can be measured. Focusing on multithreading, a prevailing technique of programming, this paper presents a new flow-sensitive technique for probabilistic pointer analysis of multithreaded programs. The proposed technique has the form of a type system and calculates the probability of every points-to relation at each program point. The key to our approach is to calculate the points-to information via a post-type derivation. The use of type systems has the advantage of associating each analysis results with a justification (proof) for the correctness of the results. This justification has the form of a type derivation and is very much required in applications like certified code.

KEYWORDS: static analysis, speculative optimizations, probabilistic alias analysis, distributed programs, semantics of multithreaded programs, type systems

INTRODUCTION

Multithreading is enjoying a growing interest and becoming a prevailing technique of programming. The use of multiple threads has several advantages: (a) concealing the delay of commands like reading from a secondary storage, (b) improving the action of programs, like web servers, that run on multiprocessors, (c) building complex systems for user interface, (d) simplifying the process of organizing huge systems of code. However, the static analysis of multithreaded programs[1–3] is intricate due to the possible interaction between multiple threads.

Among effective tools of modern programming languages are pointers, which empower coding intricate data structures. Not only does the uncertainty of pointer values at compile time complicate analysis of programs, but also retard program compilation by compelling the program optimization and analysis to be conservative. The pointer analysis[4–6] of programs is a challenging problem in which researchers have trade space and time costs for precision. However, binary decision diagrams[7] have been used to ease the difficulty of this trade off.

At any program point and for every pair of memory references, a traditional pointer analysis figures out whether one of these references may point to, definitely points to, or definitely does point to the other reference. For most of pairs of the memory references the points-to relation is of type "may be". This is specially the case for techniques that prefer speed over accuracy. Traditional optimization techniques are not robust enough to treat the cases "may be" and "definitely" differently. The idea behind speculative optimization is to subsidize the "maybe" case, specially if the probability of "maybe" cab be specifically quantified[8,9].

Pointer analysis[10,11] is among the most important program analyses of multithreaded programs. Pointer analysis of multithreaded programs has many applications; (a) mechanical binding of file operations that are in abeyance, (b) optimizations for memory systems like prefetching and relocating remote data calculations, (c) equipping compilers with necessary information for optimizations like common subexpression elimination and induction variable elimination, and

(d) relaxing the process of developing complex tools for software engineering like program slicers and race detectors.

This paper presents a new technique for pointer analysis of multithreaded programs. The proposed technique is probabilistic; it anticipates precisely for every program point the probability of every points-to relation. Building on a type system, the proposed approach is control-flow-sensitive. The key to the presented analysis is to calculate probabilities for points-to relations through the compositional use of inference rules of a type system. The proposed technique associates with every analysis a proof (type derivation) for the correctness of the analysis.

Among techniques to approach static analysis of programs is the algorithmic style. However, the proposed technique of this paper has the form of a type system. The algorithmic style does not reflect how the analysis results are obtained because it works on control-flow graphs of programs; not on phrase structures as in the case of type systems. Therefore the type-system approach[4,12–14] is perfect for applications that require to handle a justifications (proof) for correctness of analysis results together with each individual analysis. An example of such applications is certified code. What contributes to suitability of type-system tools to produce such proofs is the relative simplicity of its inference rules. This simplicity is a much appreciated property in applications that require justifications. In the type-system approach, the justifications take the form of type derivations.

Motivation

Fig. 1 presents a motivating example of our work. This example uses three pointer variables (a, b, and e) that point at two variables (c and d). We suppose that (i) the condition of the *if* statement at line 2 is true with probability 0.6, (ii) the condition of the *if* statement at line 9 is true with probability 0.5, and

```
1.  a := &c;
2.  if(...) then b := &c
3.          else b := &d;
4.  par{
5.      {a := &c}
6.      {a := &d}
7.      };
8.  while(...)
9.      if(...) then e := &d
10.             else e := 5;
```

Fig. 1 A motivating example.

Table 1 Results of pointer analysis of program in Fig. 1.

Program point	Pointer information
first point	$\{t \mapsto \emptyset \mid t \in \mathit{Var}\}$
between lines 1 & 2	$\{a \mapsto \{(c', 1)\},$ $t \mapsto \emptyset \mid x \neq t\}$
point between 3 & 4	$\{a \mapsto \{(c', 1)\},$ $b \mapsto \{(c', 0.6), (d', 0.4)\},$ $t \mapsto \emptyset \mid t \notin \{a, b\}\}$
point between 7 & 8	$\{a \mapsto \{(c', 0.5), (d', 0.5)\},$ $b \mapsto \{(c', 0.6), (d', 0.4)\},$ $t \mapsto \emptyset \mid t \notin \{a, b\}\}$
last point	$\{a \mapsto \{(c', 0.5), (d', 0.5)\},$ $e \mapsto \{(d', \frac{1}{100} \times \Sigma_{i=1}^{i=100} (\frac{1}{2})^i)\},$ $b \mapsto \{(c', 0.6), (d', 0.4)\},$ $t \mapsto \emptyset \mid t \notin \{a, b, c\}\}$

(iii) the loop at line 8 iterates at most 100 times. These statistical and probabilistic information can be obtained using edge profiling[15–18]. In absence of edge profiling, heuristics can be used. The work presented in this paper aims at introducing a probabilistic pointer analysis that produces results like that in Table 1. The aim is also to associate each such pointer-analysis result with a justification for the correctness of the result. This justification takes the form of a type derivation in our proposed technique which is based on a type system.

Contributions

Contributions of this paper are the following:

1. A new pointer analysis technique, that is probabilistic and flow-sensitive, for multithreaded programs.
2. A new probabilistic operational-semantics for multithreaded programs.

Organization

The remainder of the paper is organized in three sections as follows. The first of these sections presents a simple language equipped with parallel and pointer constructs. This section also presents a new probabilistic operational semantics for the constructs of the language that we study. The second of these sections introduces a type system to carry probabilistic pointer analysis of parallel programs. This involves introducing suitable notions for pointer types, a subtyping relation, and a detailed proof for the soundness of the proposed type system w.r.t. the semantics presented in the paper. Related work is reviewed in the last section of the paper.

$$\begin{aligned}
& n \in \mathbb{Z},\ x \in \mathit{Var},\ \mathit{and} \oplus \in \{+,-,\times\} \\
e \in \mathit{Aexprs} &::= x \mid n \mid e_1 \oplus e_2 \\
b \in \mathit{Bexprs} &::= \mathit{true} \mid \mathit{false} \mid \neg b \mid e_1 = e_2 \mid e_1 \leqslant e_2 \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \\
S \in \mathit{Stmts} &::= x := e \mid x := \&y \mid *x := e \mid x := *y \mid \mathit{skip} \mid S_1; S_2 \mid \mathit{if}\ b\ \mathit{then}\ S_t\ \mathit{else}\ S_f \mid \\
& \mathit{while}\ b\ \mathit{do}\ S_t \mid \mathit{par}\{\{S_1\},\dots,\{S_n\}\} \mid \mathit{par\text{-}if}\{(b_1,S_1),\dots,(b_n,S_n)\} \mid \mathit{par\text{-}for}\{S\}.
\end{aligned}$$

Fig. 2 The programming language.

PROBABILISTIC OPERATIONAL SEMANTICS

This section presents the programming language we study and a probabilistic pointer analysis for its constructs. We build our language (Fig. 2) on the *while* language, originally presented by Hoare in 1969, by equipping it with commands dealing with pointers and parallel computations. The parallel concepts dealt with in our language are fork-join, conditionally spawned threads, and parallel loops. These concepts are represented by the commands *par*, *par-if*, and *par-for*, respectively. States of our proposed operational semantics are defined as follows:

Definition 1 1. $\mathit{Addrs} = \{x' \mid x \in \mathit{Var}\}$ and $\mathit{Val} = \mathbb{Z} \cup \mathit{Addrs}$.
2. $\gamma \in \Gamma = \mathit{Var} \to \mathit{Val}$.
3. $\mathit{state} \in \mathit{States} = \{(\gamma, p) \mid \gamma \in \Gamma \wedge p \in [0,1]\} \cup \{\mathit{abort}\}$.

Typically, a state is a function from the set of variables to the set of values (integers). In our work, we enrich the set of values with a set of symbolic addresses and enrich each state with a probabilistic value that is meant to measure the probability with which this state is reached. The *abort* state is there to capture any case of de-reference that is unsafe; i.e., de-referencing a variable that contains no address. We assume that the set of program variables, *Var*, is finite.

Except that arithmetic and Boolean operations are not allowed on pointers, the semantics of arithmetic and Boolean expressions are defined as usual (Fig. 3). The inference rules of Fig. 4 define the transition relation $\rightsquigarrow$ of our operational semantics.

We notice that none of the assignment statements changes the probability component of a given pre-state to produce the corresponding post-state. The symbol p_{if} used in the inference rules of the *if* statement denotes a number in $[0,1]$ and measures the probability that the condition of the statement is true. This probabilistic information can be obtained using edge profiling[15–18]. In absence of edge profiling, heuristics can be used.

The *par* command is the main parallel concept. This concept is also known as cobegin-coend or fork-join. The execution of this command amounts to starting concurrently executing the threads of the command at the beginning of the construct and then to wait for the completion of these executions at the end of the construct. Then the subsequent command can be executed. The inference rule (*par-sem*) approximates the execution method of the *par* command. The probability p' in the rule (*par-sem*) is multiplied by $1/n!$ (not by $1/n$ as the reader may expect) because the permutation θ finds one of the $n!$ ways in which the threads can be sorted and then executed. As an example, the reader may consider applying the rule *par-sem* when $n = 3$ and the threads are $S_1 : a := b + c$, $S_2 : b := a \times c$, and$S_3 : c := a - b$. The semantics of *par-if* and *par-for* commands are defined using that of the *par* command.

PROBABILISTIC POINTER ANALYSIS

The purpose of a typical pointer analysis is to assign to every program point a points-to function. The domain of this function is the set of all pairs of pointers and the codomain is the set $\{$*definitely exists*, *definitely does not exist*, *may exist*$\}$. The codomain describes the points-to relation between pairs of memory references. For most of the pointer pairs, the points-to relation is "may exist". This is specially the case for techniques of pointer analysis that give priority for speed over efficiency. The common drawback for most existing program optimization techniques is that they cannot treat the "maybe" and "definitely does not exist" cases differently. Speculative optimizations are meant to overcome this disadvantage via working on the result of analyses that can measure the probability that a points-to relation exist between two pointers.

This section presents a new technique for probabilistic pointer analysis for multithreaded programs. The technique has the form of a type system and its goal is to accurately calculate the likelihood at each program point for every points-to relation. The advantages of the proposed technique include the

$$[\![n]\!]\,\gamma = n \quad [\![\&x]\!]\,\gamma = x' \quad [\![x]\!]\,\gamma = \gamma(x) \quad [\![\mathit{true}]\!]\,\gamma = \mathit{true} \quad [\![\mathit{false}]\!]\,\gamma = \mathit{false}$$

$$[\![*x]\!]\,\gamma = \begin{cases} \gamma(y) & \text{if } \gamma(x) = y', \\ ! & \text{otherwise.} \end{cases} \qquad [\![e_1 \oplus e_2]\!]\,\gamma = \begin{cases} [\![e_1]\!]\,\gamma \oplus [\![e_2]\!]\,\gamma & \text{if } [\![e_1]\!]\,\gamma, [\![e_2]\!]\,\gamma \in \mathbb{Z}, \\ ! & \text{otherwise.} \end{cases}$$

$$[\![\neg A]\!]\,\gamma = \begin{cases} \neg([\![A]\!]\,\gamma) & \text{if } [\![A]\!]\,\gamma \in \{\mathit{true}, \mathit{false}\}, \\ ! & \text{otherwise.} \end{cases} \qquad [\![e_1 = e_2]\!]\,\gamma = \begin{cases} ! & \text{if} [\![e_1]\!]\,\gamma = ! \text{ or } [\![e_2]\!]\,\gamma = !, \\ \mathit{true} & \text{if } [\![e_1]\!]\,\gamma = [\![e_2]\!]\,\gamma \neq !, \\ \mathit{false} & \text{otherwise.} \end{cases}$$

$$[\![e_1 \leqslant e_2]\!]\,\gamma = \begin{cases} ! & \text{if } [\![e_1]\!]\,\gamma \notin \mathbb{Z} \text{ or } [\![e_2]\!]\,\gamma \notin \mathbb{Z}, \\ [\![e_1]\!]\,\gamma \leqslant [\![e_2]\!]\,\gamma & \text{otherwise.} \end{cases}$$

$$\text{For } \diamond \in \{\wedge, \vee\},\ [\![b_1 \diamond b_2]\!]\,\gamma = \begin{cases} ! & \text{if } [\![b_1]\!]\,\gamma = ! \text{ or } [\![b_2]\!]\,\gamma = !, \\ [\![b_1]\!]\,\gamma \diamond [\![b_2]\!]\,\gamma & \text{otherwise.} \end{cases}$$

Fig. 3 Semantics of arithmetic and Boolean expressions.

simplicity of the inference rules of the type system and that no dependence profile information (information describing dependencies between threads) is required. Dependence profile information, required by some multithreading techniques like Ref. 19, is expensive to get. The proposed technique is flow-sensitive. The key to our technique is to calculate points-to probabilities via a post type derivation for a given program using the bottom points-to type as a pre type.

The following definition presents some notations that are used in the rest of the paper.

Definition 2 1. $\mathit{Addrs} = \{x' \mid x \in \mathit{Var}\}$ and $\mathit{Addrs}_p = \mathit{Addrs} \times [0, 1]$.

2. $\mathit{Pre\text{-}PTS} = \{pts \mid pts : \mathit{Var} \to 2^{\mathit{Addrs}_p}$ s.t. $\forall y \in \mathit{Var}.\ (y', p_1), (y', p_2) \in pts(x) \Longrightarrow p_1 = p_2\}$.

3. For $pts \in \mathit{Pre\text{-}PTS}$ and $x \in \mathit{Var}$, $\sum_{pts} x = \sum_{(z',p) \in pts(x)} p$.

4. For every $pts \in \mathit{Pre\text{-}PTS}$ and $x \in \mathit{Var}$, $A_{pts}(x) = \{z' \mid \exists p > 0.\ (z', p) \in pts(x)\}$.

5. For $A \in \mathit{Addrs}_p$, $pts \in \mathit{Pre\text{-}PTS}$, and $0 \leqslant q \leqslant 1$,

(a) $A \times q = \{(y', p \times q) \mid (y', p) \in A\}$.

(b) $pts \times q$ is the function defined by $(pts \times q)(x) = pts(x) \times q$.

We note that the set of symbolic addresses *Addrs* is enriched with probabilities to form the set Addrs_p. In line with real situations, the condition on the elements of *Pre-PTS* excludes maps that assign the same address for a variable with two different probabilities. The notation $\sum_{pts} x$ denotes the probability that the variables x has an address with respect to pts. The notation $A_{pts}(x)$ denotes the set of addresses that have a non-zero probability to get into x. The multiplication operations of Definition 2.5 are necessary to join many points-to types (each with a different probability) into one type.

A formalization for the concepts of the set of points-to types *PTS*, the subtyping relation $\leqslant$, and the relation $\models\ \subseteq\ \Gamma \times \mathit{PTS}$ are in the subsequent definition.

Definition 3 1. $PTS = \{pts \in \mathit{Pre\text{-}PTS} \mid \forall x \in \mathit{Var}.\ \sum_{pts} x \leqslant 1\}$.

2. $pts \leqslant pts' \overset{\text{def}}{\Longleftrightarrow} \forall x.\ A_{pts}(x) \subseteq A_{pts'}(x)$.

3. $pts \equiv pts' \overset{\text{def}}{\Longleftrightarrow} \forall x.\ A_{pts}(x) = A_{pts'}(x)$.

4. $(\gamma, p) \models pts \overset{\text{def}}{\Longleftrightarrow} (\forall x.\ \gamma(x) \in \mathit{Addrs} \Longrightarrow \exists q > 0.\ (\gamma(x), q) \in pts(x))$.

A way to calculate an upper bound for a set of n points-to types is introduced in the following definition.

Definition 4 Suppose $pts_1, \ldots, pts_n$ is a sequence of n points-to types and $0 \leqslant q_1, \ldots, q_n \leqslant 1$ is a sequence of n numbers whose sum is less than or equal to 1. Then $\nabla((pts_1, q_1), \ldots, (pts_n, q_n)) : \mathit{Var} \to 2^{\mathit{Addrs}_p}$ is the function defined by:

$$\nabla((pts_1, q_1), \ldots, (pts_n, q_n))(x) = \{(z', p) \mid (\exists i.\ z' \in A_{pts_i}(x)) \wedge (p = \sum_{(z', p_k) \in pts_k(x)} q_k \times p_k)\}.$$

We note that the order of the points-to lattice is the point-wise inclusion. However, probabilities are implicitly taken into account in the definition of supremum which is based on Definition 4. Letting the probabilities of points-to relations be involved in the definition of the order relation complicates the formula of calculating the lattice supremum. Besides that this complication is not desirable, introducing probabilities apparently does not improve the type system results. The definition for $(\gamma, p) \models pts$ makes sure that a variable that has an address under γ is allowed (positive probability) to contain the same address

$$\frac{\llbracket e \rrbracket\,\gamma = !}{x := e : (\gamma, p) \rightsquigarrow \textit{abort}} \qquad \frac{\llbracket e \rrbracket\,\gamma \neq !}{x := e : (\gamma, p) \rightsquigarrow (\gamma[x \mapsto \llbracket e \rrbracket\,\gamma], p)} \qquad \frac{\gamma(x) = z' \quad z := e : (\gamma, p) \rightsquigarrow \textit{state}}{*x := e : (\gamma, p) \rightsquigarrow \textit{state}}$$

$$\frac{\gamma(x) \notin \textit{Addrs}}{*x := e : (\gamma, p) \rightsquigarrow \textit{abort}} \qquad \frac{}{x := \&y : (\gamma, p) \rightsquigarrow (\gamma[x \mapsto y'], p)} \qquad \frac{\gamma(y) \notin \textit{Addrs}}{x := *y : (\gamma, p) \rightsquigarrow \textit{abort}}$$

$$\frac{\gamma(y) = z' \quad x := z : (\gamma, p) \rightsquigarrow (\gamma', p)}{x := *y : (\gamma, p) \rightsquigarrow (\gamma', p)} \qquad \frac{}{\textit{skip} : (\gamma, p) \rightsquigarrow (\gamma, p)} \qquad \frac{S_1 : (\gamma, p) \rightsquigarrow \textit{abort}}{S_1; S_2 : (\gamma, p) \rightsquigarrow \textit{abort}}$$

$$\frac{S_1 : (\gamma, p) \rightsquigarrow (\gamma'', p'') \quad S_2 : (\gamma'', p'') \rightsquigarrow \textit{state}}{S_1; S_2 : (\gamma, p) \rightsquigarrow \textit{state}} \qquad \frac{\llbracket b \rrbracket\,\gamma = !}{\textit{if } b \textit{ then } S_t \textit{ else } S_f : (\gamma, p) \rightsquigarrow \textit{abort}}$$

$$\frac{\llbracket b \rrbracket\,\gamma = \textit{true} \quad S_t : (\gamma, p) \rightsquigarrow \textit{abort}}{\textit{if } b \textit{ then } S_t \textit{ else } S_f : (\gamma, p) \rightsquigarrow \textit{abort}} \qquad \frac{\llbracket b \rrbracket\,\gamma = \textit{true} \quad S_t : (\gamma, p) \rightsquigarrow (\gamma', p')}{\textit{if } b \textit{ then } S_t \textit{ else } S_f : (\gamma, p) \rightsquigarrow (\gamma', p_{\textit{if}} \times p')}$$

$$\frac{\llbracket b \rrbracket\,\gamma = \textit{false} \quad S_f : (\gamma, p) \rightsquigarrow \textit{abort}}{\textit{if } b \textit{ then } S_t \textit{ else } S_f : (\gamma, p) \rightsquigarrow \textit{abort}} \qquad \frac{\llbracket b \rrbracket\,\gamma = \textit{false} \quad S_f : (\gamma, p) \rightsquigarrow (\gamma', p')}{\textit{if } b \textit{ then } S_t \textit{ else } S_f : (\gamma, p) \rightsquigarrow (\gamma', (1 - p_{\textit{if}}) \times p')}$$

$$\frac{\llbracket b \rrbracket\,\gamma = !}{\textit{while } b \textit{ do } S_t : (\gamma, p) \rightsquigarrow \textit{abort}} \qquad \frac{\llbracket b \rrbracket\,\gamma = \textit{false}}{\textit{while } b \textit{ do } S_t : (\gamma, p) \rightsquigarrow (\gamma, p)}$$

$$\frac{\llbracket b \rrbracket\,\gamma = \textit{true} \quad S : (\gamma, p) \rightsquigarrow \textit{abort}}{\textit{while } b \textit{ do } S_t : (\gamma, p) \rightsquigarrow \textit{abort}}$$

$$\frac{\llbracket b \rrbracket\,\gamma = \textit{true} \quad S : (\gamma, p) \rightsquigarrow (\gamma'', p'') \quad \textit{while } b \textit{ do } S_t : (\gamma'', p'') \rightsquigarrow \textit{state}}{\textit{while } b \textit{ do } S_t : (\gamma, p) \rightsquigarrow \textit{state}}$$

• **Fork-join**:

$$\frac{(\exists\, \theta : \{1, \dots, n\} \to \{1, \dots, n\}).\, S_{\theta(1)}; S_{\theta(2)}; \dots; S_{\theta(n)} : (\gamma, p) \rightsquigarrow (\gamma', p')}{\textit{par}\{\{S_1\}, \dots, \{S_n\}\} : (\gamma, p) \rightsquigarrow (\gamma', \frac{1}{n!} \times p')}\ (\textit{par-sem})$$

$$\frac{(\exists\, \theta : \{1, \dots, n\} \to \{1, \dots, n\}).\, S_{\theta(1)}; S_{\theta(2)}; \dots; S_{\theta(n)} : (\gamma, p) \rightsquigarrow \textit{abort}}{\textit{par}\{\{S_1\}, \dots, \{S_n\}\} : (\gamma, p) \rightsquigarrow \textit{abort}}$$

• **Conditionally spawned threads**:

$$\frac{\textit{par}\{\{\textit{if } b_1 \textit{ then } S_1 \textit{ else skip}\}, \dots, \{\textit{if } b_n \textit{ then } S_n \textit{ else skip}\}\} : (\gamma, p) \rightsquigarrow \textit{state}}{\textit{par-if}\{(b_1, S_1), \dots, (b_n, S_n)\} : (\gamma, p) \rightsquigarrow \textit{state}}$$

• **Parallel loops**:

$$\frac{\exists n.\ \textit{par}\{\overbrace{\{S\}, \dots, \{S\}}^{n-times}\} : (\gamma, p) \rightsquigarrow \textit{state}}{\textit{par-for}\{S\} : (\gamma, p) \rightsquigarrow \textit{state}}$$

Fig. 4 Inference rules of the semantics.

under pts. As for Definition 4, we can interpret the elements of the sequence $q_1, \dots, q_n$ as weights for the elements of the sequence $pts_1, \dots, pts_n$, respectively. Therefore the map $\nabla((pts_1, q_1), \dots, (pts_n, q_n))$ joins $pts_1, \dots, pts_n$ into one type with respect to the weights.

The following lemma proves that the upper bound of the previous definition is indeed a points-to type.

Lemma 1 *The map* $\nabla((pts_1, q_1), \dots, (pts_n, q_n))$ *of previous definition is a points-to type.*

Proof: Suppose that $\nabla((pts_1, q_1), \dots, (pts_n, q_n))(x) = \{(z'_1, t_1), (z'_2, t_2), \dots, (z'_m, t_m)\}$. To show the required we need to show that (a) $0 \leqslant t_i \leqslant 1$ and (b) $0 \leqslant \Sigma_i t_i \leqslant 1$. Since (b) implies (a), it is enough to show (b). Suppose that $\forall 1 \leqslant i \leqslant n$, $pts_i(x) =$

$$\begin{array}{c} \\ z'_1 \\ z'_2 \\ \vdots \\ z'_m \end{array} \begin{array}{c} \begin{array}{cccc} pts_1 & pts_2 & \dots & pts_n \end{array} \\ \left[\begin{array}{cccc} p_{11} & p_{12} & \dots & p_{1n} \\ p_{21} & p_{22} & \dots & p_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m1} & p_{m2} & \dots & p_{mn} \end{array}\right] \end{array} \begin{array}{c} \\ \left(\begin{array}{c} q_1 \\ q_2 \\ \vdots \\ q_n \end{array}\right) = \left(\begin{array}{c} t_1 \\ t_2 \\ \vdots \\ t_m \end{array}\right) \end{array}$$

Fig. 5 A matrix multiplication needed in the proof of Lemma 1.

$\{(z'_1, p_{1i}), (z'_2, p_{2i}), \dots, (z'_m, p_{mi})\}$, where $\forall 1 \leqslant j \leqslant m,\ p_{ji} = 0$ if $z_j \notin A_{pts_i}(x)$. Then according to Definition 4 the values $t_1, \dots, t_m$ can be equivalently calculated by the matrix multiplication of Fig. 5.

Then:

$$\begin{aligned} \Sigma_i\, t_i &= (\Sigma_i\, q_i \times p_{1i}) + (\Sigma_i\, q_i \times p_{2i}) + \dots \\ &\quad + (\Sigma_i\, q_i \times p_{mi}) \\ &= (q_1 \times \Sigma_i\, p_{i1}) + (q_2 \times \Sigma_i\, p_{i2}) + \dots \\ &\quad + (q_n \times \Sigma_i\, p_{in}). \end{aligned}$$

We note that $\forall j,\ 0 \leqslant \Sigma_i\, p_{ij} \leqslant 1$ by definition of pts_j and $\forall j,\ 0 \leqslant q_j \leqslant 1$. Therefore this last summation is less than 1. □

Lemma 2 *Suppose that $A = \{pts_1, \dots, pts_n\} \subseteq$ PTS and $pts = \nabla((pts_1, 1/n), \dots, (pts_n, 1/n))$. Then with respect to definitions of ∇, the subtyping, and equality relations introduced in Definitions 3.2, 3.3, and 4, respectively, the set PTS is a complete lattice where $\vee A = pts$.*

Proof: Clearly pts is an upper bound for A. Moreover for every x, $A_{pts}(x) = \cup_i A_{pts_i}(x)$. Therefore pts is the least upper bound of A. □

The inference rules of our proposed type system for probabilistic pointer analysis are shown in Fig. 6.

The judgement of an arithmetic expression has the form $e : pts \to A$. The intuition (Lemma 3) of this judgement is that any address that e evaluates to in a state of type pts is included in the set A as the second component of a pair whose first component is a non-zero probability. The judgement for a statement S has the form $S : pts \to pts'$ and guarantees that if the execution of S in a state of type pts terminates then the reached state is of type pts'. This is proved in Theorem 1.

Concerning the inference rules, some comments are in order. In the rule ($:= *^{prob}$), since there are n possible ways to modify x, the post-type is calculated from the pre-type by assigning x its value according to the upper bound of the n ways. The upper bound is consider to enable the analysis to cover all possible executions of the statement. In the rule ($* :=^{prob}$), there are n variables, $\{z_1, \dots, z_n\}$, that have a chance of getting modified. This produces n post-types in the pre conditions of the rule. Therefore the post-type is calculated from the pre-type by assigning each of the n variables its image under the upper bound of the n post-types. In the rule (if^{prob}), p is the probability that the condition of the *if* statement is true. The rule (par^{prob}) has this form in order for the analysis result of any thread S_i of the *par* statement to consider the fact that any other thread may have been executed before the thread in hand. As it is the case in the operational semantics, the rules for conditionally spawned threads ($par\text{-}if^{prob}$) and parallel loops ($par\text{-}for^{prob}$) are built on the rule (par^{prob}). In the following we give an example for the application of the rule (par^{prob}). Let:

- S_1 : *if* b_1 *then* $x := \&y$ *else* $x := 5$,
- $S_2 : x := \&z$;
- S_{par} : $par\{\{S_1\}, \{S_2\}\}$,
- $pts = \{t \mapsto \emptyset \mid t \in Var\}$, $pts_1 = \{x \mapsto \{(y', 0.4)\}, t \mapsto \emptyset \mid x \neq t \in Var\}$, and $pts_2 = \{x \mapsto \{(z', 1)\}, t \mapsto \emptyset \mid x \neq t \in Var\}$.

We suppose that the condition b_1 in S_1 succeeds with probability 0.4. Then we have the following:

$$\begin{aligned} &\nabla((pts, 1/2), (pts_1, 1/2)) = \\ &\qquad \{x \mapsto \{(y', 0.25)\}, t \mapsto \emptyset \mid x \neq t \in Var\}, \\ &\nabla((pts, 1/2), (pts_2, 1/2)) = \\ &\qquad \{x \mapsto \{(z', 0.5)\}, t \mapsto \emptyset \mid x \neq t \in Var\}, \text{ and} \\ &\nabla((pts_1, 1/2), (pts_2, 1/2)) = \\ &\{x \mapsto \{(y', 0.25), (z', 0.5)\}, t \mapsto \emptyset \mid x \neq t \in Var\}. \end{aligned}$$

Clearly, $S_1 : \nabla((pts, 1/2), (pts_2, 1/2)) \to pts_1$ and $S_2 : \nabla((pts, 1/2), (pts_1, 1/2)) \to pts_2$. These two judgements constitute the hypotheses for the rule (par^{prob}). Therefore using the rule (par^{prob}), we can conclude that $S_{par} : pts \to \nabla((pts_1, 1/2), (pts_2, 1/2))$. The post type of S_{par} clearly covers all semantics states that can be reached by executing S_{par}. Now we give an example for the application of the rule ($par\text{-}if^{prob}$). Let:

- $S_1 : x := \&y$,
- $S_2 : x := \&z$,

$$\frac{}{n : pts \to \emptyset} \quad \frac{}{x : pts \to pts(x)} \quad \frac{}{e_1 \oplus e_2 : pts \to \emptyset} \quad \frac{e : pts \to A}{x := e : pts \to pts[x \mapsto A]}\,(:=^{prob})$$

$$\frac{pts(y) = \{(z'_1, p_1), \dots, (z'_n, p_n)\} \quad \forall i.\, x := z_i : pts \to pts_i}{x := *y : pts \to pts[x \mapsto \nabla((pts_1, p_1), \dots, (pts_n, p_n))(x)]}\,(:= *^{prob}) \quad \frac{}{skip : pts \to pts}$$

$$\frac{pts(x) = \{(z'_1, p_1), \dots, (z'_n, p_n)\} \quad \forall z'_i \in A_{pts}(x).\, z_i := e : pts \to pts_i}{*x := e : pts \to pts[z_i \mapsto \nabla((pts, 1-p_i), (pts_i, p_i))(z_i) \mid z'_i \in A_{pts}(x)]}\,(* :=^{prob})$$

$$\frac{}{x := \&y : pts \to pts[x \mapsto \{(y', 1)\}]}\,(:= \&^{prob}) \quad \frac{S_1 : pts \to pts'' \quad S_2 : pts'' \to pts'}{S_1; S_2 : pts \to pts'}\,(seq^{prob})$$

$$\frac{S_t : pts \to pts_t \quad S_f : pts \to pts_f}{\textit{if } b \textit{ then } S_t \textit{ else } S_f : pts \to \nabla((pts_t, p), (pts_f, 1-p))}\,(if^{prob})$$

$$\frac{S_i : \nabla\{(pts, 1/n), (pts_j, 1/n) \mid j \neq i\} \to pts_i}{par\{\{S_1\}, \dots, \{S_n\}\} : pts \to \nabla((pts_1, 1/n), \dots, (pts_n, 1/n))}\,(par^{prob})$$

$$\frac{par\{\{\textit{if } b_1 \textit{ then } S_1 \textit{ else skip}\}, \dots, \{\textit{if } b_n \textit{ then } S_n \textit{ else skip}\}\} : pts \to pts'}{par\text{-}if\{(b_1, S_1), \dots, (b_n, S_n)\} : pts \to pts'}\,(par\text{-}if^{prob})$$

$$\frac{\forall n.\, par\{\overbrace{\{S\}, \dots, \{S\}}^{n-times}\} : pts \to pts'}{par\text{-}for\{S\} : pts \to pts'}\,(par\text{-}for^{prob}) \quad \frac{n = 0}{\textit{while } b \textit{ do } S_t : pts \to pts}\,(whl_1^{prob})$$

$$\frac{n \geqslant 1 \qquad \forall 1 \leqslant i \leqslant n.\, S_t : pts \to^i pts_i}{\textit{while } b \textit{ do } S_t : pts \to \nabla((pts_1, 1/n), \dots, (pts_n, 1/n))}\,(whl_2^{prob})$$

$$\frac{pts'_1 \leqslant pts_1 \quad S : pts_1 \to pts_2 \quad pts_2 \leqslant pts'_2}{S : pts'_1 \to pts'_2}\,(csq^{prob})$$

Fig. 6 The inference rules for the type system for probabilistic pointer analysis.

- $S_{par\text{-}if}$: $par\text{-}if\{(b_1, S_1), (true, S_2)\}$, and
- $pts' = \{x \mapsto \{(y', 0.25), (z', 0.5)\}, t \mapsto \emptyset \mid x \neq t \in Var\}$ and $pts = \{t \mapsto \emptyset \mid t \in Var\}$.

We suppose that the condition b_1 succeeds with probability 0.4. By the previous example it should be clear that $par\{\{\textit{if } b_1 \textit{ then } S_1 \textit{ else skip}\},\{\textit{if true then } S_2 \textit{ else skip}\}\} : pts \to pts'$. This last judgement constitutes the hypothesis for the rule ($par\text{-}if^{prob}$). Therefore using the rule ($par\text{-}if^{prob}$), we can conclude that $S_{par\text{-}if} : pts \to pts'$. The post type of $S_{par\text{-}if}$ clearly covers all semantics states that can be reached by executing $S_{par\text{-}if}$. In rules (whl_1^{prob}) and (whl_2^{prob}), n represents an upper bound for the trip-count of the loop. The post-type of (whl_2^{prob}) is an upper bound for post-types resulting for all number of iterations bounded by n.

The proof of the following lemma is straightforward.

Lemma 3 *1.* $pts \leqslant pts' \Longrightarrow (\forall(\gamma, p).(\gamma, p) \models pts \Longrightarrow (\gamma, p) \models pts')$

2. Suppose $e : pts \to A$ *and* $(\gamma, p) \models pts$*. Then* $[\![e]\!]\gamma \in Addrs$ *implies* $([\![e]\!]\gamma, q) \in A$*, for some* $q > 0$*.*

Lemma 3.1 formalizes the soundness of points-to types. Lemma 3.2 shows that for a certain state that is of a certain type, if the evaluation of an expression with respect to the state is an address, then this evaluation is surely (positive probability) approximated by the evaluation of the expression with respect to the type.

The following theorem proves the soundness of the type system. The meant soundness implies that the type system respects the operational semantics with respect to the relation $\models$ whose definition is based on probabilities.

Theorem 1 (*Soundness*) *Suppose that* $S : pts \to pts'$, $S : (\gamma, p) \rightsquigarrow (\gamma', p')$*, and* $(\gamma, p) \models pts$*. Then* $(\gamma', p') \models pts'$*.*

Proof: A structure induction on type derivation can be used to complete the proof of this theorem. Some cases are presented below.

- The case of ($:=^{prob}$): in this case $p' =$

p, $pts' = pts[x \mapsto A]$, and $\gamma' = \gamma[x \mapsto [\![e]\!]\gamma]$. Hence by Lemma 3.2, $\gamma \models (pts, p)$ implies $\gamma' \models (pts', p')$.

- The case of $(:= *^{prob})$: in this case for some $z \in Var$, $\gamma(y) = z'$ and $x := z : (\gamma, p) \rightsquigarrow (\gamma', p)$. For some i, $z' = z'_i$ since $(\gamma, p) \models pts$. Hence by assumption $x := z_i : pts \to pts_i$. Therefore by soundness of $(:=^{prob})$, $(\gamma', p) \models pts_i \leqslant pts' = pts[x \mapsto \nabla((pts_1, p_1), \ldots, (pts_n, p_n))(x)]$.
- The case of $(* :=^{prob})$: in this case there exists $z \in Var$ such that $\gamma(x) = z'$ and $z := e : (\gamma, p) \rightsquigarrow (\gamma', p)$. For some i, $z' = z'_i$ since $(\gamma, p) \models pts$. Hence by assumption $z_i := e : pts \to pts_i$. Therefore by soundness of $(:=^{prob})$, $(\gamma', p) \models pts_i \leqslant pts' = pts[z_i \mapsto \nabla((pts, 1 - p_i), (pts_i, p_i))(z_i) \mid z'_i \in A_{pts}(x)]$.
- The case of (par^{prob}): in this case there exists a permutation $\theta : \{1, \ldots, n\} \to \{1, \ldots, n\}$ and $n + 1$ states $(\gamma_1, p_1), \ldots, (\gamma_{n+1}, p_{n+1})$ such that $(\gamma, p) = (\gamma_1, p_1)$, $(\gamma', p') = (\gamma_{n+1}, (1/n!) \times p'_{n+1})$, and for every $1 \leqslant i \leqslant n$, $S_{\theta(i)} : (\gamma_i, p_i) \to (\gamma_{i+1}, p_{i+1})$. Also $(\gamma_1, p_1) \models pts \leqslant \nabla\{(pts, 1/n), (pts_j, 1/n) \mid j \neq 1\}$. Therefore by the induction hypothesis $(\gamma_2, p_2) \models pts_1 \leqslant \nabla\{(pts, 1/n), (pts_j, 1/n) \mid j \neq 2\}$. Again by the induction hypothesis we get $(\gamma_3, p_3) \models pts_2$. Therefore by a simple induction on n, we can show that $(\gamma_{n+1}, p_{n+1}) \models pts_n \leqslant \nabla((pts_1, 1/n), \ldots, (pts_n, 1/n)) = pts'$. This implies $(\gamma', p') = (\gamma_{n+1}, (1/n!) \times p'_{n+1}) \models pts'$
- The case of $(par\text{-}for^{prob})$: in this case there exists n such that

$$par\{\overbrace{\{S\}, \ldots, \{S\}}^{n-times}\} : (\gamma, p) \rightsquigarrow (\gamma', p').$$

By induction hypothesis we have

$$par\{\overbrace{\{S\}, \ldots, \{S\}}^{n-times}\} : pts \to pts'.$$

Therefore by the soundness of (par^{prob}), $(\gamma', p') \models pts'$.

- The case of (whl_2^{prob}): in this case there exist $m \leqslant n$ and $m+1$ states, $(\gamma_1, p_1), \ldots, (\gamma_{m+1}, p_{m+1})$, such that $(\gamma, p) = (\gamma_1, p_1)$, $(\gamma', p') = (\gamma_{m+1}, p_{m+1})$, and $\forall 1 \leqslant i \leqslant m.\ S : (\gamma_i, p_i) \rightsquigarrow (\gamma_{i+1}, p_{i+1})$. By induction hypothesis we have $(\gamma', p') \models pts_m \leqslant \nabla((pts_1, 1/n), \ldots, (pts_n, 1/n))$. Therefore $(\gamma', p') \models pts'$ as required. □

We note that probabilities are mentioned implicitly in Theorem 1. This is in the condition that $(\gamma, p) \models pts$. Some of the implications of this implicit consideration of probabilities are explicit in Lemma 3.2. As an example for the theorem, executing the statement S_{par}, defined above, from the semantics state $\gamma = \{t \mapsto 0 \mid t \in Var\}$ may result in the state $\gamma' = \{t \mapsto 0, x \mapsto z' \mid x \neq t \in Var\}$. This happens if S_2 is executed after S_1. Clearly we have that $\gamma \models \{t \mapsto \emptyset \mid t \in Var\}$ and $\gamma' \models \{x \mapsto \{(y', 0.25), (z', 0.5)\}, t \mapsto \emptyset \mid x \neq t \in Var\}$.

One source of attraction in the use of type systems for program analysis is the relative simplicity of the inference rules. This simplicity is very important when practical implementation is concerned. The simplicity of the rules naturally simplifies implementations of rules and hence the type system. In particular, from experience related to coding similar type systems, we believe that the implementation of the type system presented in this paper is straightforward and efficient in terms of space and time.

RELATED WORK

Analysis of multithreaded programs:

Typically, analyses of multithreaded programs are classified into two main categories: (a) techniques that were originally designed for sequential programs and later extended to analyse multithreaded programs and (b) techniques that were designed specifically for analysing, optimizing, or correcting multithreaded programs.

The first category includes flow-insensitive approaches providing an easy way to analyse multithreaded programs. This is done via considering all possible combinations of statements used in a parallel structure. The drawback of this approach is that it is not practical enough due to the huge number of combinations. However, flow-sensitive approaches of sequential programs were also extended to cover multithreaded programs. Examples of these techniques are constant propagation[20], code motion[21], and reaching definitions[22].

The category of techniques that were designed specifically for multithreaded programs include deadlock detection, data race detection, and weak memory consistency. A round abeyance to gain resources usually results in a deadlock situation[3,23,24]. Synchronization analysis is a typical start to study deadlock detection for multithreaded programs. In absence of synchronization, if two parallel threads write to the same memory location, a situation of a data race[2] results. Data race analyses aim at eliminating data race situations as they are mainly programmer error. Models of weak memory consistency[1] aim at improving performance of hardware. This improvement usually results in complicating parallel programs construction and analysis.

Probabilistic pointer analysis and speculative optimizations:

Although pointer analysis is a well-established program analysis and many techniques have been suggested, there is no single technique that is believed to be the best choice[25]. The trade-off between accuracy and time costs hinders a universal pointer analysis and motivates application-directed techniques for pointer analysis[26]. A probabilistic pointer analysis that is flow-sensitive and context-insensitive has been presented for Java programs[27]. While our work is based on type systems, previous work is based on interprocedural control flow graphs whose edges are enriched with probabilities. While our work treats multithreaded programs, the work in Ref. 27 treats only sequential programs. Context-sensitive and control-flow-sensitive pointer analyses[4,10,28,29] are known to be accurate but not scalable. On the other hand the context-insensitive control-flow-insensitive techniques[6,11] are scalable but excessively conservative. A convenient mixture of accuracy and scalability is introduced by some technique[7,30,31] to optimize the trade-off mentioned above. The probabilistic pointer analysis of a simple imperative language[8,9] and the pointer analysis of multithreaded programs[32] have been studied. However, none of these typical techniques for pointer analysis study the probabilistic pointer analysis of multithreaded programs.

Speculative optimizations[33–36] are considered by many program analyses. A probabilistic technique for memory disambiguation was proposed[34]. This technique measures the probability that two array references alias. Nevertheless this approach is not convenient to pointers. By lessening the safety of analysis, a pointer analysis that considers speculation was introduced[35]. Another unsafe analysis, which achieves scalability using transfer functions, was proposed[36]. The problem with these last two approaches is that they do not compute the probability information required by speculative optimizations.

Type systems in program analysis:

There are general algorithms[4,13,14,37–39] for using type systems to present data flow analyses, which are monotone and forward or backward. While a way[14,37] to reason about program pairs using relational Hoare logic exists, program optimizations[14,38] as types systems also exist. Type systems were also used to cast safety policies for resource usage, information flow, and carrying-code abstraction[40,41]. Proving the soundness of compiler optimizations for imperative languages, using type systems, gained much interest[12–14] of many researchers. Other work studies translating proofs of functional correctness using wp-calculus[42] and using a Hoare logic[14]. There are other optimizations[43] that boost program quality besides maintaining program semantics.

Edge and path profiling:

Edge (path) profiling research simply aims at profiling programs edges (paths). The profiling process can be done statically or dynamically. Profiling techniques can be classified into:

- Sample-based techniques[16,17] which profile representative parts of active edges and paths,
- One-time profiling methods which profile only part of the execution of the program to cut down the overhead[17,44],
- Instrumentation-based techniques[45] which are more convenient for programs with comparably anticipated behaviour, and
- Hardware profiling which employs hardware to gather edge profiles using existing hardware for branch anticipation[18].

Using a parallel data-flow diagram[46], many of these techniques are applicable to the language studied in this paper. In particular, a hybrid sampling and instrumentation approach[15] is a convenient choice giving its simplicity and powerful.

Acknowledgements: This work was started during the author's sabbatical at Institute of Cybernetics, Estonia in the year 2009. The author is grateful to T. Uustalu for fruitful discussions. This work was partially supported by the EU FP6 IST project MOBIUS. The author is also indebted to the anonymous reviewers whose queries and comments improved the paper.

REFERENCES

1. Gelado I, Cabezas J, Navarro N, Stone JE, Patel SJ, Hwu WW (2010) An asymmetric distributed shared memory model for heterogeneous parallel systems. In: Hoe JC, Adve VS (eds) *ASPLOS*, ACM, pp 347–58.
2. Leung K, Huang Z, Huang Q, Werstein P (2009) Maotai 2.0: Data race prevention in view-oriented parallel programming. In: *PDCAT*, IEEE Computer Society, pp 263–71.
3. Xiao X, Lee JJ (2010) A true o(1) parallel deadlock detection algorithm for single-unit resource systems and its hardware implementation. *IEEE Trans Parallel Distr Syst* **21**, 4–19.
4. El-Zawawy MA (2011) Program optimization based pointer analysis and live stack-heap analysis. *Int J Comput Sci Issues* **8**, 98–107.

5. El-Zawawy MA (2011) Flow sensitive-insensitive pointer analysis based memory safety for multithreaded programs. In: Murgante B, Gervasi O, Iglesias A, Taniar D, Apduhan BO (eds) *ICCSA (5)*, Springer, vol 6786 of *Lecture Notes in Computer Science*, pp 355–69.
6. Adams S, Ball T, Das M, Lerner S, Rajamani SK, Seigle M, Weimer W (2002) Speeding up dataflow analysis using flow-insensitive pointer analysis. In: Hermenegildo MV, Puebla G (eds) *SAS*, Springer, vol 2477 of *Lecture Notes in Computer Science*, pp 230–46.
7. Berndl M, Lhoták O, Qian F, Hendren LJ, Umanee N (2003) Points-to analysis using bdds. In: *PLDI*, ACM, pp 103–14.
8. Chen PS, Hwang YS, Ju RDC, Lee JK (2004) Interprocedural probabilistic pointer analysis. *IEEE Trans Parallel Distr Syst* **15**, 893–907.
9. Silva JD, Steffan JG (2006) A probabilistic pointer analysis for speculative optimizations. In: Shen JP, Martonosi M (eds) *ASPLOS*, ACM, pp 416–25.
10. Yu H, Xue J, Huo W, Feng X, Zhang Z (2010) Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In: Moshovos A, Steffan JG, Hazelwood KM, Kaeli DR (eds) *CGO*, ACM, pp 218–29.
11. Anderson P, Binkley D, Rosay G, Teitelbaum T (2002) Flow insensitive points-to sets. *Inform Software Tech* **44**, 743–54.
12. Bertot Y, Grégoire B, Leroy X (2004) A structured approach to proving compiler optimizations based on dataflow analysis. In: Filliatre JC, Paulin Mohring C, Werner B (eds) *TYPES*, Springer, vol 3839 of *Lecture Notes in Computer Science*, pp 66–81.
13. Laud P, Uustalu T, Vene V (2006) Type systems equivalent to data-flow analyses for imperative languages. *Theor Comput Sci* **364**, 292–310.
14. Saabas A, Uustalu T (2008) Program and proof optimizations with type systems. *J Logic Algebr Program* **77**, 131–54.
15. Bond MD, McKinley KS (2005) Continuous path and edge profiling. In: *MICRO*, IEEE Computer Society, pp 130–40.
16. Anderson JAM, Berc LM, Dean J, Ghemawat S, Henzinger MR, Leung ST, Sites RL, Vandevoorde MT, et al (1997) Continuous profiling: Where have all the cycles gone? *ACM Trans Comput Syst* **15**, 357–90.
17. Suganuma T, Yasue T, Kawahito M, Komatsu H, Nakatani T (2005) Design and evaluation of dynamic optimizations for a java just-in-time compiler. *ACM Trans Program Lang Syst* **27**, 732–85.
18. Vaswani K, Thazhuthaveetil MJ, Srikant YN (2005) A programmable hardware path profiler. In: *CGO*, IEEE Computer Society, pp 217–28.
19. Steffan JG, Colohan CB, Zhai A,Mowry TC (2005) The stampede approach to thread-level speculation. *ACM Trans Comput Syst* **23**, 253–300.
20. Lee J, Midkiff SP, Padua DA (1997) Concurrent static single assignment form and constant propagation for explicitly parallel programs. In: Li Z, Yew PC, Chatterjee S, Huang CH, Sadayappan P, Sehr DC (eds) *LCPC*, Springer, vol 1366 of *Lecture Notes in Computer Science*, pp 114–30.
21. Knoop J, Steffen B (1999) Code motion for explicitly parallel programs. In: *PPOPP*, pp 13–24.
22. Sarkar V (2009) Challenges in code optimization of parallel programs. In: de Moor O, Schwartzbach MI (eds) *CC*, Springer, vol 5501 of *Lecture Notes in Computer Science*, p 1.
23. Kim BC, Jun SW, Hwang DJ, Jun YK (2009) Visualizing potential deadlocks in multithreaded programs. In: Malyshkin V (ed) *PaCT*, Springer, vol 5698 of *Lecture Notes in Computer Science*, pp 321–30.
24. Wang Y, Kelly T, Kudlur M, Lafortune S, Mahlke SA (2008) Gadara: Dynamic deadlock avoidance for multithreaded programs. In: Draves R, van Renesse R (eds) *OSDI*, USENIX Association, pp 281–94.
25. Hind M, Pioli A (2000) Which pointer analysis should I use? In: *ISSTA*, pp 113–23.
26. Hind M (2001) Pointer analysis: haven't we solved this problem yet? In: *PASTE*, pp 54–61.
27. Sun Q, Zhao J, Chen Y (2011) Probabilistic points-to analysis for java. In: Knoop J (ed) *CC*, Springer, vol 6601 of *Lecture Notes in Computer Science*, pp 62–81.
28. Hardekopf B, Lin C (2009) Semi-sparse flow-sensitive pointer analysis. In: Shao Z, Pierce BC (eds) *POPL*, ACM, pp 226–38.
29. Wang J, Ma X, Dong W, Xu HF, Liu W (2009) Demand-driven memory leak detection based on flow- and context-sensitive pointer analysis. *J Comput Sci Tech* **24**, 347–56.
30. Whaley J, Lam MS (2004) Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Pugh and Chambers[47], pp 131–44.
31. Zhu J, Calman S (2004) Symbolic pointer analysis revisited. In: Pugh and Chambers[47], pp 145–57.
32. Rugina R, Rinard MC (2003) Pointer analysis for structured parallel programs. *ACM Trans Program Lang Syst* **25**, 70–116.
33. Ramalingam G (1996) Data flow frequency analysis. In: *PLDI*, pp 267–77.
34. Ju RDC, Collard JF, Oukbir K (1999) Probabilistic memory disambiguation and its application to data speculation. *Comput Architect News* **27**, 27–30.
35. Fernández M, Espasa R (2002) Speculative alias analysis for executable code. In: *IEEE PACT*, IEEE Computer Society, pp 222–31.
36. Bhowmik A, Franklin M (2003) A fast approximate interprocedural analysis for speculative multithreading compilers. In: Banerjee U, Gallivan K, Gonzalez A (eds) *ICS*, ACM, pp 32–41.
37. Benton N (2004) Simple relational correctness proofs for static analyses and program transformations. In: Jones ND, Leroy X (eds) *POPL*, ACM, pp 14–25.

38. Nielson HR, Nielson F (2002) Flow logic: A multi-paradigmatic approach to static analysis. In: Mogensen TAE, Schmidt DA, Sudborough IH (eds) *The Essence of Computation*, Springer, vol 2566 of *Lecture Notes in Computer Science*, pp 223–44.
39. Nicola RD, Gorla D, Hansen RR, Nielson F, Nielson HR, Probst CW, Pugliese R (2010) From flow logic to static type systems for coordination languages. *Sci Comput Program* **75**, 376–97.
40. Beringer L, Hofmann M, Momigliano A, Shkaravska O (2004) Automatic certification of heap consumption. In: Baader F, Voronkov A (eds) *LPAR*, Springer, vol 3452 of *Lecture Notes in Computer Science*, pp 347–62.
41. Besson F, Jensen TP, Pichardie D (2006) Proof-carrying code fromcertified abstract interpretation and fixpoint compression. *Theor Comput Sci* **364**, 273–91.
42. Barthe G, Grégoire B, Kunz C, Rezk T (2009) Certificate translation for optimizing compilers. *ACM Trans Program Lang Syst* **31**, Article 18.
43. Aspinall D, Beringer L, Momigliano A (2007) Optimisation validation. *Electron Notes Theor Comput Sci* **176**, 37–59.
44. Zilles CB, Sohi GS (2002) Master/slave speculative parallelization. In: *MICRO*, ACM/IEEE, pp 85–96.
45. Joshi R, Bond MD, Zilles CB (2004) Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems. In: *CGO*, IEEE Computer Society, pp 239–50.
46. Grunwald D, Srinivasan H (1993) Data flow equations for explicitly parallel programs. In: *PPOPP*, pp 159–68.
47. Pugh W, Chambers C (eds) (2004) *Proceedings of the ACMSIGPLAN 2004 Conference on Programming Language Design and Implementation 2004*, Washington DC, USA, June 9–11, 2004, ACM.