Flow Sensitive-Insensitive Pointer Analysis Based Memory Safety for Multithreaded Programs

Mohamed A. El-Zawawy

Department of Mathematics
Faculty of Science
Cairo University
Giza 12613
Egypt
maelzawawy@cu.edu.eg

Abstract. The competency of pointer analysis is crucial for many compiler optimizations, transformations, and checks like memory safety. The potential interaction between threads in multithreaded programs complicates their pointer analysis and memory-safety check. The trade-off between accuracy and scalability remains a main issue when studying these analyses. In this work, we present novel approaches for the pointer analysis and memory safety of multithreaded programs as simply structured type systems.

In order to balance accuracy and scalability, the type system proposed for pointer analysis of multithreaded programs is flow-sensitive and it invokes another flow-insensitive type system for parallel constructs. Therefore the proposed pointer analysis is described as flow sensitive-insensitive. The third type system presented in this paper takes care of memory safety of multithreaded programs and is an extension of the type system of pointer analysis. Programs having types in memory-safety type system, are guaranteed to be memory safe. Type derivations serve as proofs for correctness of the result of every pointer analysis and memory-safety check. Such proofs are required in the area of proof-carrying code.

Keywords: Pointer analysis, memory safety, operational semantics, multithreaded programs, type systems.

1 Introduction

Two facts contribute to the enormous importance that the pointer analysis of multithreaded programs enjoys. One fact is that pointer information is important for many compiler optimizations and corrections [23]. The other fact is the growing interest in multithreading as a mainstream practice of programming. One important use of pointer information is to statically judge the memory safety of programs. That is to reason about (a) the existence of pointer arithmetics when they are not allowed by the syntax of the language and (b) the existence of dangling pointers (de-referencing of variables that contain no pointers). The fact that pointer analysis is a main tool in code parallelization magnifies importance of pointer analysis.

For multithreaded programs, compilation and program analyses are challenging problems [17,23] because the potential interaction between various threads creates difficulty in extending techniques of compiling and analyzing sequential programs to cover multithreaded ones. For the pointer analysis, the interaction happens when a thread writes a pointer variable that is simultaneously accessed by another thread. Such interactions result in enlarging sets of pointers that variables may point to. For the sake of correctness, analyses of multithreaded programs must conveniently interpret the interaction between various threads.

Pointer analysis [7,9] calculates information about memory locations that are pointed to by program pointers. The memory safety analysis aims at statically proves that the program does not treat pointers illegally according to the language syntax. One way to classify techniques of pointer analysis and memory safety is according to flow-sensitivity of approaches i.e. into flow-sensitive and flow-insensitive. The approaches of flow-insensitive neglect the program's flow of control. Hence in these techniques the program statements are dealt with as if they are executable any number of times in any probable order. Flow-insensitive approaches [1,2] are believed to be less precise and more efficient than flow sensitive ones. Therefore there is a trade-off between using flow-sensitive and flow-insensitive approaches. In this paper, we introduce pointer analysis and memory safety techniques for multithreaded programs. Aiming at capturing the advantages of the two approaches, the proposed techniques mix flow sensitivity and insensitivity.

Typically, static analysis of programs is done in an algorithmic style through which algorithms work on control-flow graphs of programs rather than on their syntactic structures. The algorithmic style has the drawback of working like a black box in the sense that it is not clear how the analysis was done. Therefore in applications like proof-carrying code or certified code, where a proof for the correctness of analysis results is required to be delivered with the result, the algorithmic style is not an ideal choice. Type systems have established themselves as good tools to carry static analysis instead of algorithms specially for applications of proof-carrying code [3,14,20]. Type rules are relatively easy to interpret and type derivations are good format for required proofs. The techniques presented in this paper for pointer analysis and memory safety of multithreaded programs are in the form of type systems.

Figure 1 presents the programming language that we study. The language is the simple *while* language [11] enriched with commands for structured parallel constructs and pointer manipulations. Join-fork constructs, conditionally spawned threads, and parallel loops are parallel constructs included in the language. The join-fork (*par*) construct begins the execution of its threads concurrently at the beginning of the construct and

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

Fig. 1. The programming language

then waits for the accomplishment of these executions at the end of the *par* construct. The parallel loop construct, *par-for*, executes in parallel a statically unknown number of threads that have the same code (the loop body). The construct including conditionally spawned threads is that of *par-if* and it executes in parallel its n threads where the execution of thread (b_i, S_i) includes the execution of S_i only if b_i is true.

This paper presents a new technique for pointer analysis of multithreaded programs. Type systems are basic tools of our proposed technique which mixes concepts of flow sensitive and insensitive analyses. More precisely, the technique has the form of a type system that is flow-sensitive and that invokes another type system which is flow-insensitive when parallel constructs are encountered. The paper also presents another type system that checks memory-safety of multithreaded programs and that utilizes the pointer information obtained by our pointer analysis. We show that if a program has types in the memory safety type system then it is memory-safe in the sense that it is guaranteed not to abort due to illegal pointer operations.

Motivation

```
1.  x := &y;
2.  *x := 2;
3.  par{
4.      {y := &x}
5.      {y := 5;
6.       *x := &z}
7.      };
8.  x := &z;
9.  z := 2;
10.  y := *z
```

Program point	Pointer information
first point	$\{t \mapsto \emptyset \mid t \in Var\}$
between lines 1 & 2	$\{x \mapsto \{y\}, t \mapsto \emptyset \mid x \neq t\}$
points between 2 & 8	$\{x \mapsto \{y\}, y \mapsto \{x,z\}, t \mapsto \emptyset \mid t \notin \{x,y\}\}$
points between 8 & 10	$\{x \mapsto \{z\}, y \mapsto \{x,z\}, t \mapsto \emptyset \mid t \notin \{x,y\}\}$
last point	$\{x \mapsto \{z\}, t \mapsto \emptyset \mid x \neq t\}$

Fig. 2. A motivating example together with its pointer analysis

The program in Figure 2 is a motivating example of the work presented in this paper. We note that this program aborts because the command at the last line de-references the variable z that has no address. Therefore this program is not memory safe. It is the task of this paper to introduce a technique that statically (without running programs) tests memory safety of multithreaded programs like this one. The information that z has no addresses can be inferred from the result of a pointer analysis for the program. Hence the paper introduces an efficient flow sensitive-insensitive pointer analysis whose results for our example program are in the table of Figure 2.

Contributions

Contributions of this paper are the following:

1. A new flow sensitive-insensitive pointer analysis technique for multithreaded programs.
2. An original type system for flow-insensitive pointer analysis of multithreaded programs.
3. An original analysis that checks memory safety of multithreaded programs.

Organization

The rest of the paper is organized in sections that present the following topics in the following order:

1. Related work.
2. The proposed technique (type systems) for pointer analysis of multithreaded programs.
3. The proposed technique (type system) for memory safety of multithreaded programs.
4. Operational semantics of our language (Figure 1).
5. Conclusion.

2 Related Work

Pointer Analysis and Memory Safety

The pointer analysis and memory safety for sequential programs have been studied extensively for decades [7,10,9,25,6]. Flow-sensitive pointer analyses [8,26,30], which are more natural to most applications, consider the order of program commands. Mostly these analyses perform an abstract interpretation of program using dataflow analysis to associate each program point with a points-to relation. Flow-insensitive pointer analyses [1,2] do not consider the order of program commands. Typically the output of these analyses, which are performed using a constraint-based approach, is a points-to relation that is valid all over the program. Clearly the flow-sensitive approach is more precise but less efficient than the flow-insensitive one. The work [25] presents a type and effect analysis for detecting memory and type errors in C source code.

Although problems of pointer analysis and memory safety for sequential programs were studied extensively, a little effort was done towards a pointer and a memory-safety analyses for multithreaded programs. In [22], a flow sensitive analysis for multithreaded programs was introduced. This analysis associates each program point with a triple of points-to relations. This in turn complicates the the analysis and creates a sort of redundancy in the collected points-to information. The work in [7] presents a design for a formal low-level multi-threaded language enriched with region-based memory management and synchronization constructs. Well-typed programs of this language are claimed to be memory safe and race free. Investigating the details of these approaches and our work makes it apparent that our work is simpler and more accurate than these approaches. Moreover our approach provides a proof for the correctness of the pointer analysis and memory safety for each program. To the best of our knowledge, such proof is not known to be provided by any other existing approach.

Type Systems in Program Analysis

The work in [3,14,20,6,19] is among the closest work to ours in the sense that it uses type systems to achieve the program analysis in a way similar to ours. The work in [14] shows that a good deal of program analysis can be done using type systems. More precisely, it proves that for every analysis in a certain class of data-flow analyses, there exists a type system such that a program checks with a type if and only if the type is a supertype for the set resulting from running the analysis on the program. The type system in [18] and the flow-logic work in [20], which is used in [19] to study security of the coordinated systems, are very similar to [14]. For the simple while language, the work in [3] introduces type systems for constant folding and dead code elimination and also logically proves correctness of optimizations. Safety policies for information flow, carrying-code abstraction, and resource usage were also casted using type systems [4,5]. Earlier, related work (with structurally-complex type systems) is [21].

To the best of our knowledge, our approach is the first attempt to use type systems to check memory safety based pointer analysis for multithreaded programs and associates every individual check with a justification for correctness.

Analysis of Multithreaded Programs

The analysis of multithreaded programs is a challenging area [23] that receives growing interest; threading complicates the program analysis. There are several directions of research in this area [15,13]. Deadlock which results from round waiting to gain resources is a problem of multithreading computing. Researchers have developed various techniques for deadlock detection [12,27,28]. The aim of the analysis of synchronization constructs [24,29] is to explore how the synchronization actions apart executions of program segments. The result of this analysis can be used by compiler to appropriately add join-fork constructs. Data race describes the situation when a memory location is accessed by two threads (one of them writes in the location) without synchronization. One direction of research focuses on data race detection [16].

The problem with almost all the work refereed to above is that it does not apply to pointer programs. More precisely, for some of the work the application is possible only if we have the result of a pointer analysis for the input pointer program. The techniques presented in this paper have the advantage of being simpler and more reliable than the techniques refereed to above that would work in the presence of a pointer analysis.

3 Pointer Analysis

This section presents a new technique for pointer analysis of multithreaded programs that allow shared pointers to be updated simultaneously. Our technique is basically a flow-sensitive analysis that invokes flow-insensitive one for the analysis of parallel constructs; join-fork constructs, parallel loops, and conditionally spawned threads. Therefore the technique can be described as flow sensitive-insensitive. Both of the analyses take the form of compositional type systems that are simply structured. The pointer information calculated by the analyses have the form of types assigned to expressions and statements. The correctness of collected pointer information is proved by type derivations. Hence the presented analyses proceed by assigning a type to each program point of a statement (program). A points-to type associates each variable in the program with a conservative approximation of the addresses that may get into the variable. The soundness of all type systems presented in this paper is proved using the operational semantics presented in Section 5. The set of states of this semantics is denoted by Γ.

The following definition introduces the set of points-to types *PTS* and the relation $\models \subseteq \Gamma \times PTS$:

Definition 1. *1.* $Addrs = \{x' \mid x \in Var\}$.
2. $PTS = \{pts \mid pts : Var \rightarrow 2^{Addrs}\}$.
3. $pts \leq pts' \overset{\text{def}}{\iff} \forall x \in Var.\ pts(x) \subseteq pts'(x)$.
4. $\gamma \models pts \overset{\text{def}}{\iff} (\forall x \in Var.\ \gamma(x) \in Addrs \implies \gamma(x) \in pts(x))$.

We start with introducing a type system for flow-insensitive pointer analysis of multithreaded programs. Then a type system for flow-sensitive analysis that invokes the flow-insensitive type system is introduced.

3.1 Flow-Insensitive Pointer Analysis

The types, *PTS*, of our type system for flow-insensitive pointer analysis are introduced in the previous definition. The inference rules of the type system are the following:

$$\frac{}{x : pts, pts(x)} \qquad \frac{}{n : pts, \emptyset} \qquad \frac{}{e_1 \oplus e_2 : pts, \emptyset} \qquad \frac{}{skip : pts} \qquad \frac{e : pts, A \quad A \subseteq pts(x)}{x := e : pts}\ (:=^p_{insen})$$

$$\frac{y' \in pts(x)}{x := \&y : pts}\ (:= \&^p_{insen}) \qquad \frac{\forall z' \in pts(x).\ z := e : pts}{*x := e : pts}\ (* :=^p_{insen})$$

$$\frac{\forall z' \in pts(y).\ x := z : pts}{x := *y : pts}\ (:= *^p_{insen}) \qquad \frac{\forall i.\ S_i : pts}{par\{\{S_1\}, \ldots, \{S_n\}\} : pts}\ (par^p_{insen})$$

$$\frac{\forall i.\ S_i : pts}{par\text{-}if\{(b_1, S_1), \ldots, (b_n, S_n)\} : pts}\ (par\text{-}if^p_{insen}) \qquad \frac{S : pts}{par\text{-}for\{S\} : pts}\ (par\text{-}for^p_{insen})$$

$$\frac{S_1 : pts \quad S_2 : pts}{S_1; S_2 : pts}\ (seq^p_{insen}) \qquad \frac{S_t : pts \quad S_f : pts}{if\ b\ then\ S_t\ else\ S_f : pts}\ (if^p_{insen})$$

$$\frac{S_t : pts}{while\ b\ do\ S_t : pts}\ (whl^p_{insen}) \qquad \frac{S : pts \quad pts \leq pts'}{S : pts'}\ (csq^p_{insen})$$

The judgements of an expression e and a statement S have forms $e : pts, A$ and $S : pts$, respectively. As formalized by Lemma 1, A is the collection of addresses that e may evaluate to in a state of type pts. The judgement of S guarantees that if the execution of S in a state of type pts terminates in a state γ', then γ' has type pts.

The inference rule $(:=^p_{insen})$ says that for the assignment statement to be of type pts, the set $pts(x)$ has to cover the set of addresses A. The other inference rules corresponding to other assignment commands are clarified similarly. A type invariant is necessary to type statements like *while*, *par*, *par-if*, and *par-for*. A fix-point algorithm can be used to find type invariants. The monotonicity of rules of the type system and the fact that the set of points-to types *PTS* is a complete lattice guarantee the convergence of the algorithm.

Lemma 1. *1. Suppose* $e : pts, A$ *and* $\gamma \models pts$. *Then* $[\![e]\!]\gamma \in Addrs$ *implies* $[\![e]\!]\gamma \in A$.
2. $pts \leq pts' \iff (\forall \gamma.\ \gamma \models pts \implies \gamma \models pts')$.

Proof. (1) is obvious. The right-to-left direction of (2) is proved as follows. Suppose $y' \in pts(x)$. Then the state $\{(x, y'), (t, 0) \mid t \in Var \setminus \{x\}\}$ is of type pts and hence of type pts' implying that $y' \in pts'(x)$. Therefore $pts(x) \subseteq pts'(x)$. Since x is arbitrary, $pts \leq pts'$. The other direction is easy.

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

Proof. The proof is by structure induction on the type derivation. We demonstrate some cases.

- The case of $(:=^p_{insen})$: in this case $\gamma' = \gamma[x \mapsto [\![e]\!]\gamma]$. Therefore by the previous lemma $\gamma \models pts$ implies $\gamma' \models pts$.
- The case of $(* :=^p_{insen})$: in this case there exists $z \in Var$ such that $\gamma(x) = z'$ and $z := e : \gamma \rightsquigarrow \gamma'$. Because $\gamma \models pts$, $z' \in pts(x)$ and hence by assumption $z := e : pts$. Therefore by soundness of $(:=^p_{insen})$, $\gamma' \models pts$.
- The case of (par^p_{insen}): in this case there exist a permutation $\theta : \{1, \ldots, n\} \to \{1, \ldots, n\}$ and $n + 1$ states $\gamma = \gamma_1, \ldots, \gamma_{n+1} = \gamma'$ such that for every $1 \leq i \leq n$, $S_{\theta(i)} : \gamma_i \to \gamma_{i+1}$. Also $\gamma_1 \models pts$. Therefore by the induction hypothesis $\gamma_2 \models pts$. Again by the induction hypothesis we get $\gamma_3 \models pts$. Therefore by a simple induction on n, we can show that $\gamma' = \gamma_{n+1} \models pts$.
- The case of $(par\text{-}for^p_{insen})$: in this case there exists n such that $par\{\overbrace{\{S\}, \ldots, \{S\}}^{n-times}\} : \gamma \rightsquigarrow \gamma'$. By induction hypothesis we have $S : pts$. By (par^p_{insen}) we conclude that $par\{\overbrace{\{S\}, \ldots, \{S\}}^{n-times}\} : pts$. Therefore by the soundness of (par^p_{insen}), $\gamma' \models pts'$.

3.2 Flow Sensitive-Insensitive Pointer Analysis

This section presents the basic type system that carries the pointer analysis of multi-threaded programs. For parallel constructs the type system calls the flow-insensitive type system presented in the previous subsection. Therefore the type system presented here is described as flow sensitive-insensitive analysis. The following are the rules of the type system:

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

$$\frac{}{x := \&y : pts \to pts[x \mapsto \{y'\}]}\ (:= \&^p_{sen}) \qquad \frac{}{skip : pts \to pts}$$

$$\frac{\forall z' \in pts(y).\ x := z : pts \to pts'}{x := *y : pts \to pts'}\ (:= *^p_{sen}) \quad \frac{\forall z' \in pts(x).\ z := e : pts \to pts'}{*x := e : pts \to pts'}\ (* :=^p_{sen})$$

$$\frac{par\{\{S_1\},\ldots,\{S_n\}\} : pts}{par\{\{S_1\},\ldots,\{S_n\}\} : pts \rightarrow pts}\ (par^p) \qquad \frac{S_1 : pts \rightarrow pts'' \quad S_2 : pts'' \rightarrow pts'}{S_1; S_2 : pts \rightarrow pts'}\ (seq^p_{sen})$$

$$\frac{par\text{-}if\{(b_1, S_1),\ldots,(b_n, S_n)\} : pts}{par\text{-}if\{(b_1, S_1),\ldots,(b_n, S_n)\} : pts \rightarrow pts}\ (par\text{-}if^p_{sen})$$

$$\frac{par\text{-}for\{S\} : pts}{par\text{-}for\{S\} : pts \rightarrow pts}\ (par\text{-}for^p_{sen}) \qquad \frac{S_t : pts \rightarrow pts' \quad S_f : pts \rightarrow pts'}{if\ b\ then\ S_t\ else\ S_f : pts \rightarrow pts'}\ (if^p_{sen})$$

$$\frac{S_t : pts \rightarrow pts}{while\ b\ do\ S_t : pts \rightarrow pts}\ (whl^p_{sen}) \qquad \frac{pts'_1 \le pts_1 \quad S : pts_1 \rightarrow pts_2 \quad pts_2 \le pts'_2}{S : pts'_1 \rightarrow pts'_2}\ (csq^p_{sen})$$

The judgements of an expression e and a statement S have forms $e : pts \rightarrow A$ and $S : pts \rightarrow pts'$, respectively. The intuition of these judgements are similar to that described in the previous section for corresponding judgments. A typical pointer analysis for a program S takes the form of a post-type derivation starting with the bottom type (mapping variables to $\emptyset$) as the pre-type.

Lemma 2. *Suppose $e : pts \rightarrow A$ and $\gamma \models pts$. Then $[\![e]\!]\gamma \in Addrs$ implies $[\![e]\!]\gamma \in A$.*

Theorem 2. (*Soundness*) *Suppose that $S : pts \rightarrow pts'$, $S : \gamma \rightsquigarrow \gamma'$, and $\gamma \models pts$. Then $\gamma' \models pts'$.*

Proof. The proof is by structure induction on the type derivation. Some cases are shown below.

- The case of $(:=^p_{sen})$: in this case $pts' = pts[x \mapsto A]$ and $\gamma' = \gamma[x \mapsto [\![e]\!]\gamma]$. Therefore by the previous lemma $\gamma \models pts$ implies $\gamma' \models pts'$.
- The case of $(* :=^p_{sen})$: in this case there exists $z \in Var$ such that $\gamma(x) = z'$ and $z := e : \gamma \rightsquigarrow \gamma'$. Because $\gamma \models pts$, $z' \in pts(x)$ and hence by assumption $z := e : pts \rightarrow pts'$. Therefore by soundness of $(:=^p_{sen})$, $\gamma' \models pts'$.
- The cases of (par^p_{sen}), $(par\text{-}for^p_{sen})$, and $(par\text{-}if^p_{sen})$ follow directly from soundness of the type system of the previous subsection (Theorem 1).

4 Memory Safety

This section presents a new technique that statically studies the memory safety of multithreaded programs. A memory safe program is one that is guaranteed not to attempt any illegal operations with pointers like dereferencing a variable that has no pointer. The proposed technique is a type system that extends the type system of pointer analysis presented in the previous section. The extension takes the form of another type component added to points-to types. The new component is meant to capture for each program point the set of variables that must contain addresses. The resulting types of the memory-safety type system can be seen as a refitment of the points-to types. This is reflected by the fact that the pointer information collected by the pointer analysis is used in the rules of memory-safety type system.

The following definition introduces the set of safety-types MS and the relation $\models \subseteq \Gamma \times MS$:

Definition 2. – *A safety type is a pair of a points-to type pts and a subset* $v \subseteq Var$.

– $(pts, v) \leq (pts', v') \overset{\text{def}}{\Longleftrightarrow} pts \leq pts'$ *and* $v \supseteq v'$.

– *A state* γ *has type* (pts, v)*, denoted by* $\gamma \models (pts, v)$*, if* $\gamma \models pts$ *and* $\forall x \in v.\ \gamma(x) \in Addrs$.

Inference rules of our type system for memory safety are the following:

$$\frac{y \in v}{y : (x, pts, v) \curvearrowright v \cup \{x\}}\ (y_1^m) \qquad \frac{y \notin v}{y : (x, pts, v) \curvearrowright v \setminus \{x\}}\ (y_2^m) \qquad \frac{}{n : (x, pts, v) \curvearrowright v \setminus \{x\}}\ (n^m)$$

$$\frac{\forall y \in FV(e_1 \oplus e_2).\ pts(y) = \emptyset}{e_1 \oplus e_2 : (x, pts, v) \curvearrowright v \setminus \{x\}}\ (\oplus^m) \qquad \frac{x := e : pts \longrightarrow pts' \quad e : (x, pts, v) \curvearrowright v'}{x := e : (pts, v) \curvearrowright (pts', v')}\ (:=^m)$$

$$\frac{}{skip : (pts, v) \curvearrowright (pts, v)} \qquad \frac{x \in v \quad \forall z' \in pts(x)(z := e : (pts, v) \curvearrowright (pts', v'))}{*x := e : (pts, v) \curvearrowright (pts', v')}\ (* :=^m)$$

$$\frac{y \in v \quad \forall z' \in pts(y)(x := z : (pts, v) \curvearrowright (pts', v'))}{x := *y : (pts, v) \curvearrowright (pts', v')}\ (:= *^m)$$

$$\frac{x := \&y : pts \longrightarrow pts'}{x := \&y : (pts, v) \curvearrowright (pts', v \cup \{x\})}\ (:= \&^m) \qquad \frac{S : (pts, v \cap v') \curvearrowright (pts, v')}{par\text{-}for\{S\} : (pts, v) \curvearrowright (pts, v')}\ (par\text{-}for^m)$$

$$\frac{S_i : (pts, v \cap \cap_{j \neq i} v_j) \curvearrowright (pts, v_i)}{par\{\{S_1\}, \ldots, \{S_n\}\} : (pts, v) \curvearrowright (pts, \cap_i v_i)}\ (par^m) \qquad \frac{S_t : (pts, v) \curvearrowright (pts, v)}{while\ b\ do\ S_t : (pts, v) \curvearrowright (pts, v)}\ (whl^m)$$

$$\frac{S_1 : (pts, v) \curvearrowright (pts'', v'') \quad S_2 : (pts'', v'') \curvearrowright (pts', v')}{S_1; S_2 : (pts, v) \curvearrowright (pts', v')}\ (seq^m)$$

$$\frac{par\{\{if\ b_1\ then\ S_1\ else\ skip\}, \ldots, \{if\ b_n\ then\ S_n\ else\ skip\}\} : (pts, v) \curvearrowright (pts, v')}{par\text{-}if\{(b_1, S_1), \ldots, (b_n, S_n)\} : (pts, v) \curvearrowright (pts, v')}\ (par\text{-}if^m)$$

$$\frac{S_t : (pts, v) \curvearrowright (pts', v') \quad S_f : (pts, v) \curvearrowright (pts', v')}{if\ b\ then\ S_t\ else\ S_f : (pts, v) \curvearrowright (pts', v')}\ (if^m)$$

$$\frac{(pts_1', v_1') \leq (pts_1, v_1) \quad S : (pts_1, v_1) \curvearrowright (pts_2, v_2) \quad (pts_2, v_2) \leq (pts_2', v_2')}{S : (pts_1', v_1') \curvearrowright (pts_2', v_2')}\ (csq^m)$$

The judgement of an expression e has the form $e : (x, pts, v) \curvearrowright v'$. We note that for a type (pts, v) and a variable x the post type v' does not always exist. Therefore for a type (pts, v) and a variable x such a judgement does not always exist. When exists the judgement guarantees that the statement $x := e$ does not abort when executed in a state of type (pts, v) and that if the execution terminates in a state γ', then v' contains variables that contain addresses according to γ'. The judgement of a statement S has the form $S : (pts, v) \curvearrowright (pts', v')$. For a given statement S and a pre-type (pts, v) such a judgement does not always exist. This judgement, if exists, guarantees that S does not abort at any state of type (pts, v) and if the execution of S in a state of type (pts, v) terminates in a state γ', then γ' has type (pts', v'). A typical memory safety analysis for a program S takes the form of a post-type derivation starting with the bottom type (where *pts* maps variables to $\emptyset$ and $v = \emptyset$) as the pre-type.

The inference rule ($\oplus^m$) reflects that in order for the assignment $x := e_1 \oplus e_2$ to succeed both of the operands e_1 and e_2 have to be guaranteed not to be pointers. By Lemma 1 this is guaranteed if $\forall y \in FV(e_1 \oplus e_2)(pts(y) = \emptyset)$. In this case x is assigned a number therefore gets removed from v to produce v'. In the rules ($* :=^m$) and ($:= *^m$) the variables x and y, respectively, have to belong to v to guarantee that the dereferenced variables x and y do contain addresses before execution. For the rule (par^m), according to semantics of the join-fork command, *par*, one possibility is that the execution of a specific thread S_i starts before the execution of any other thread starts. Another possibility is that the execution starts after executions of all other threads end. Of course there are many other possibilities in between. Consequently, the analysis of the thread S_i must consider all such possibilities. This is reflected in the pre-type of S_i and the post-type of the *par* command. Similar explanations clarify the rules ($par\text{-}if^m$) and ($par\text{-}for^m$).

We note that a type invariant is required to type a *while* statement. Also to achieve the analysis for one of the *par*'s threads we need to know the analysis results for all other threads. However obtaining these results requires the result of analyzing the first thread. Therefore there is a kind of circularity in rule (par^m). Similar situations are in rules ($par\text{-}if^m$) and ($par\text{-}for^m$). Such issues can be treated using a fix-point algorithm. The convergence of this algorithm is guaranteed as the rules of our type system are monotone and the set of points-to types *PTS* is a complete lattice.

Lemma 3. *1.* $(pts, v) \leq (pts', v') \Longrightarrow (\forall \gamma \in \Gamma.\ \gamma \models (pts, v) \Longrightarrow \gamma \models (pts', v'))$.
2. Suppose $e : (x, pts, v) \curvearrowright v'$ *and* $\gamma \models (pts, v)$. *Then*
(a) $[\![e]\!] \neq !$, *and*
(b) If $x := e : \gamma \rightarrow \gamma'$, *then* $\forall y \in Var.\ (\gamma'(y) \in Addrs \Longrightarrow y \in v')$.

Proof. The proof of the first item is easy. The proof of the second item is by induction on the structure of type derivation:

- The case of the rule (y_1^m): in this case $\gamma' = \gamma[x \mapsto \gamma(y)]$ and $v' = v \cup \{x\}$. Since $y \in v$, $\gamma(y) \in Addrs$. Therefore $\gamma'(x) \in Addrs$ which justifies adding x to v.
- The case of the rule (y_2^m): in this case $\gamma' = \gamma[x \mapsto \gamma(y)]$ and $v' = v \setminus \{x\}$. Since $y \notin v$, $\gamma(y)$ is not guaranteed to be an address. Consequently $\gamma'(x)$ is not guaranteed to be an address which justifies removing x from v.
- The case of the rule ($\oplus^m$): in this case $\gamma' = \gamma[x \mapsto [\![e_1 \oplus e_2]\!]\gamma]$ and $v' = v \setminus \{x\}$. $[\![e_1 \oplus e_2]\!] \in \mathbb{Z}$ because $\forall y \in FV(e_1 \oplus e_2)(pts(y) = \emptyset)$, which guarantees that $\forall y \in FV(e_1 \oplus e_2).\ \gamma(y) \in \mathbb{Z}$. Consequently $\gamma'(x) \in \mathbb{Z}$ which justifies removing x from v.

Theorem 3. *(soundness and memory safety) Suppose* $S : (pts, v) \curvearrowright (pts', v')$ *and* $\gamma \models (pts, v)$. *Then*

1. *S does not abort at* γ *i.e.* $S : \gamma \not\rightsquigarrow abort$.
2. *If* $S : \gamma \rightsquigarrow \gamma'$ *then* $\gamma' \models (pts', v')$.

Proof. The proof is by structure induction on the type derivation. Some cases are shown as follows:

- The case of $(:=^m)$: this case results from the previous lemma and the soundness of pointer analysis.
- The case of $(* :=^m)$: in this case there exists $z \in Var$ such that $\gamma(x) = z'$ because $x \in v$. We have $z' \in pts(x)$ because $\gamma \models pts$. By induction hypothesis $z := e$ does not abort at γ and consequently neither does $*x := e$. For (2) we have $z := e : \gamma \rightsquigarrow \gamma'$. By assumption we have, $z := e : (pts, v) \curvearrowright (pts', v')$. Therefore by soundness of $(:=^m)$, $\gamma' \models (pts', v')$.
- The case of (par^m): (1) We outline the proof that executing the n threads in any order starting from a state γ of type (pts, v) does not abort. Suppose that $\theta : \{1, \ldots, n\} \to \{1, \ldots, n\}$ is a permutation. $\gamma \models (pts, v)$ implies $\gamma \models (pts, v \cap \cap_{j \neq \theta(1)} v_j)$. Therefore by induction hypothesis $S_{\theta(1)}$ does not abort at γ. Then either $S_{\theta(1)}$ enters an infinite loop at γ or the execution terminates at a state γ_2 which is by induction hypothesis of type $(pts, v_{\theta(1)})$. Therefore γ_2 is of type $(pts, v \cap \cap_{j \neq \theta(2)} v_j)$. Hence a simple induction on n shows (1).
  (2) In this case there exist a permutation $\theta : \{1, \ldots, n\} \to \{1, \ldots, n\}$ and $n+1$ states $\gamma = \gamma_1, \ldots, \gamma_{n+1} = \gamma'$ such that for every $1 \leq i \leq n,\ S_{\theta(i)} : \gamma_i \to \gamma_{i+1}$. Also $\gamma_1 \models (pts, v)$ implies $\gamma_1 \models (pts, v \cap \cap_{j \neq \theta(1)} v_j)$. Therefore by the induction hypothesis $\gamma_2 \models (pts, v_{\theta(1)})$. This implies $\gamma_2 \models (pts, v \cap \cap_{j \neq \theta(2)} v_j)$. Again by the induction hypothesis we get $\gamma_3 \models (pts, v_{\theta(2)})$. Therefore by a simple induction on n, we can show that $\gamma' = \gamma_{n+1} \models (pts, v_{\theta(n)})$ which implies $\gamma' \models (pts', v') = (pts, \cap_j v_j)$.
- The case of $(par\text{-}for^m)$: (1) Similarly to the previous case it is easy to show that the *par-for* command does not abort at any state of type (pts, v).
  (2) In this case there exists n such that $par\{\overbrace{\{S\}, \ldots, \{S\}}^{n-times}\} : \gamma \rightsquigarrow \gamma'$. By induction hypothesis we have $S : (pts, v \cap v') \curvearrowright (pts, v')$. By (par^m) we conclude that $par\{\overbrace{\{S\}, \ldots, \{S\}}^{n-times}\} : (pts, v) \curvearrowright (pts, v')$. Therefore by the soundness of (par^m), $\gamma' \models (pts', v')$.

5 Operational Semantics

This section presents an operational semantics for constructs of the language (Figure 1) we study. The semantics of the *par* command can be approximately interpreted as if the threads are executed sequentially in an arbitrary order. By definitions, the semantics of *par-for* and *par-if* are expressed using that of the *par* construct. Introducing operation semantics is one way to describe the meanings of the constructs of our programming language, including the parallel constructs. This is equivalent to defining a transition relation $\rightsquigarrow$ between states which are defined as follows.

Definition 3. *1. $Addrs = \{x' \mid x \in Var\}$ and $Val = \mathbb{Z} \cup Addrs$.*
2. A state is either an abort or a map $\gamma \in \Gamma = Var \longrightarrow Val$.

Rather than that arithmetic and Boolean operations are not allowed on pointers, the semantics of arithmetic and Boolean expressions are defined as usual.

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

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

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

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

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

The inference rules of our semantics (transition relation) are defined as follows:

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

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

$$\frac{\gamma(y) \notin \mathit{Addrs}}{x := *y : \gamma \rightsquigarrow \mathit{abort}} \qquad \frac{}{\mathit{skip} : \gamma \rightsquigarrow \gamma} \qquad \frac{S_1 : \gamma \rightsquigarrow \mathit{abort}}{S_1; S_2 : \gamma \rightsquigarrow \mathit{abort}} \qquad \frac{S_1 : \gamma \rightsquigarrow \gamma'' \quad S_2 : \gamma'' \rightsquigarrow \mathit{state}}{S_1; S_2 : \gamma \rightsquigarrow \mathit{state}}$$

$$\frac{[\![b]\!]\gamma = !}{\mathit{if}\ b\ \mathit{then}\ S_t\ \mathit{else}\ S_f : \gamma \rightsquigarrow \mathit{abort}} \qquad \frac{[\![b]\!]\gamma = \mathit{true} \quad S_t : \gamma \rightsquigarrow \mathit{state}}{\mathit{if}\ b\ \mathit{then}\ S_t\ \mathit{else}\ S_f : \gamma \rightsquigarrow \mathit{state}}$$

$$\frac{[\![b]\!]\gamma = \mathit{false} \quad S_f : \gamma \rightsquigarrow \mathit{state}}{\mathit{if}\ b\ \mathit{then}\ S_t\ \mathit{else}\ S_f : \gamma \rightsquigarrow \mathit{state}} \qquad \frac{[\![b]\!]\gamma = !}{\mathit{while}\ b\ \mathit{do}\ S_t : \gamma \rightsquigarrow \mathit{abort}} \qquad \frac{[\![b]\!]\gamma = \mathit{false}}{\mathit{while}\ b\ \mathit{do}\ S_t : \gamma \rightsquigarrow \gamma}$$

$$\frac{[\![b]\!]\gamma = \mathit{true} \quad S : \gamma \rightsquigarrow \gamma'' \quad \mathit{while}\ b\ \mathit{do}\ S_t : \gamma'' \rightsquigarrow \mathit{state}}{\mathit{while}\ b\ \mathit{do}\ S_t : \gamma \rightsquigarrow \mathit{state}} \qquad \frac{[\![b]\!]\gamma = \mathit{true} \quad S : \gamma \rightsquigarrow \mathit{abort}}{\mathit{while}\ b\ \mathit{do}\ S_t : \gamma \rightsquigarrow \mathit{abort}}$$

- **Join-Fork:**

$$\frac{}{\mathit{par}\{\{S_1\}, \dots, \{S_n\}\} : \gamma \rightsquigarrow \gamma'}\ \dagger \qquad \frac{}{\mathit{par}\{\{S_1\}, \dots, \{S_n\}\} : \gamma \rightsquigarrow \mathit{abort}}\ \ddagger$$

† there exist a permutation $\theta : \{1, \dots, n\} \to \{1, \dots, n\}$ and $n+1$ states $\gamma = \gamma_1, \dots, \gamma_{n+1} = \gamma'$ such that for every $1 \leq i \leq n$, $S_{\theta(i)} : \gamma_i \to \gamma_{i+1}$.

‡ there exist m such that $1 \leq m \leq n$, a one-to-one map $\beta : \{1, \dots, m\} \to \{1, \dots, n\}$, and $m+1$ states $\gamma = \gamma_1, \dots, \gamma_{m+1} = \mathit{abort}$ such that for every $1 \leq i \leq m$, $S_{\beta(i)} : \gamma_i \to \gamma_{i+1}$.

- **Conditionally Spawned Threads:**

$$\frac{\mathit{par}\{\{\mathit{if}\ b_1\ \mathit{then}\ S_1\ \mathit{else}\ \mathit{skip}\}, \dots, \{\mathit{if}\ b_n\ \mathit{then}\ S_n\ \mathit{else}\ \mathit{skip}\}\} : \gamma \rightsquigarrow \gamma'}{\mathit{par\text{-}if}\{(b_1, S_1), \dots, (b_n, S_n)\} : \gamma \rightsquigarrow \gamma'}$$

$$\frac{\mathit{par}\{\{\mathit{if}\ b_1\ \mathit{then}\ S_1\ \mathit{else}\ \mathit{skip}\}, \dots, \{\mathit{if}\ b_n\ \mathit{then}\ S_n\ \mathit{else}\ \mathit{skip}\}\} : \gamma \rightsquigarrow \mathit{abort}}{\mathit{par\text{-}if}\{(b_1, S_1), \dots, (b_n, S_n)\} : \gamma \rightsquigarrow \mathit{abort}}$$

- **Parallel Loops:**

$$\frac{\exists n.\, par\{\overbrace{\{S\},\ldots,\{S\}}^{n-times}\} : \gamma \rightsquigarrow \gamma'}{par\text{-}for\{S\} : \gamma \rightsquigarrow \gamma'} \qquad \frac{\exists n.\, par\{\overbrace{\{S\},\ldots,\{S\}}^{n-times}\} : \gamma \rightsquigarrow abort}{par\text{-}for\{S\} : \gamma \rightsquigarrow abort}$$

Some comments on the inference rules are in order. The execution of assignment command (:=) aborts at a state only if the semantics of the expression e at that state includes arithmetics on pointers. The semantics of the indirect assignment commands ($*$:= and := $*$) uses that of direct assignment (:=) and has one more source of abortion which happens due to de-referencing that is unsafe. One source for abortion of *if* and *while* statements is the un-computability of Boolean conditions of these statements which happens when a Boolean operation is tried on pointers. The execution of *par* command, the main parallel command, amounts to starting at the begin of construct the concurrent execution of all the command threads and waiting at the end of the construct for the termination of these executions. This is approximated by the inference rules for *par* command. The semantics of the other parallel commands (*par-if* and *par-for*) is defined by means of the inference rules for *par* command.

6 Conclusion

Many compiler optimizations, transformations, and checks like memory safety are directly affected by the efficiency of the crucial program analysis of pointer analysis. One factor that complicates the pointer analysis and memory-safety check of multithreaded programs is the potential interaction between threads. A main issue when studying these analyses for programs is the trade-off between accuracy and scalability. Novel approaches, in the form of simply structured type systems, for the pointer analysis and memory safety of multithreaded programs are presented in this paper.

For the sake of balancing accuracy and scalability, a flow-insensitive type system for parallel constructs is invoked by the main flow-sensitive type system proposed for pointer analysis of multithreaded programs. Hence the proposed technique is classified as flow sensitive-insensitive. This paper extends the proposed type system for pointer analysis to introduce the third type system of the paper which takes care of memory safety of multithreaded programs. Memory safe is guaranteed for programs typed in the proposed memory-safety type system. In the proposed techniques, the result of every pointer analysis and memory-safety check is associated with a correctness proof which has the form of a type derivation. These proofs are necessary in many applications like proof-carrying code (certified code).

References

1. Adams, S., Ball, T., Das, M., Lerner, S., Rajamani, S.K., Seigle, M., Weimer, W.: Speeding up dataflow analysis using flow-insensitive pointer analysis. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, p. 230. Springer, Heidelberg (2002)
2. Anderson, P., Binkley, D., Rosay, G., Teitelbaum, T.: Flow insensitive points-to sets. Information & Software Technology 44(13), 743–754 (2002)

3. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Jones, N.D., Leroy, X. (eds.) POPL, pp. 14–25. ACM, New York (2004)
4. Beringer, L., Hofmann, M., Momigliano, A., Shkaravska, O.: Automatic certification of heap consumption. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 347–362. Springer, Heidelberg (2005)
5. Besson, F., Jensen, T.P., Pichardie, D.: Proof-carrying code from certified abstract interpretation and fixpoint compression. Theor. Comput. Sci. 364(3), 273–291 (2006)
6. El-Zawawy, M.A.: Program optimization based pointer analysis and live stack-heap analysis. International Journal of Computer Science Issues 8(2), 98–107 (2011)
7. Gerakios, P., Papaspyrou, N., Sagonas, K.F.: Race-free and memory-safe multithreading: design and implementation in cyclone. In: Kennedy, A., Benton, N. (eds.) TLDI, pp. 15–26. ACM, New York (2010)
8. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. In: Shao, Z., Pierce, B.C. (eds.) POPL, pp. 226–238. ACM, New York (2009)
9. Hind, M.: Pointer analysis: haven't we solved this problem yet? In: PASTE, pp. 54–61 (2001)
10. Hind, M., Pioli, A.: Which pointer analysis should i use? In: ISSTA, pp. 113–123 (2000)
11. Hoare, C.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)
12. Kim, B.-C., Jun, S.-W., Hwang, D.J., Jun, Y.-K.: Visualizing potential deadlocks in multithreaded programs. In: Malyshkin, V. (ed.) PaCT 2009. LNCS, vol. 5698, pp. 321–330. Springer, Heidelberg (2009)
13. Knoop, J., Steffen, B.: Code motion for explicitly parallel programs. In: PPOPP, pp. 13–24 (1999)
14. Laud, P., Uustalu, T., Vene, V.: Type systems equivalent to data-flow analyses for imperative languages. Theor. Comput. Sci. 364(3), 292–310 (2006)
15. Lee, J., Midkiff, S.P., Padua, D.A.: Concurrent static single assignment form and constant propagation for explicitly parallel programs. In: Huang, C.-H., Sadayappan, P., Sehr, D. (eds.) LCPC 1997. LNCS, vol. 1366, pp. 114–130. Springer, Heidelberg (1998)
16. Leung, K.-Y., Huang, Z., Huang, Q., Werstein, P.: Maotai 2.0: Data race prevention in view-oriented parallel programming. In: PDCAT, pp. 263–271. IEEE Computer Society, Los Alamitos (2009)
17. Midkiff, S.P., Padua, D.A.: Issues in the optimization of parallel programs. In: Padua, D.A. (ed.) ICPP (2), pp. 105–113. Pennsylvania State University Press (1990)
18. Naik, M., Palsberg, J.: A type system equivalent to a model checker. ACM Trans. Program. Lang. Syst. 30(5), 1–24 (2008)
19. Nicola, R.D., Gorla, D., Hansen, R.R., Nielson, F., Nielson, H.R., Probst, C.W., Pugliese, R.: From flow logic to static type systems for coordination languages. Sci. Comput. Program. 75(6), 376–397 (2010)
20. Riis Nielson, H., Nielson, F.: Flow logic: A multi-paradigmatic approach to static analysis. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 223–244. Springer, Heidelberg (2002)
21. Palsberg, J., O'Keefe, P.: A type system equivalent to flow analysis. ACM Trans. Program. Lang. Syst. 17(4), 576–599 (1995)
22. Rugina, R., Rinard, M.C.: Pointer analysis for structured parallel programs. ACM Trans. Program. Lang. Syst. 25(1), 70–116 (2003)
23. Sarkar, V.: Challenges in code optimization of parallel programs. In: de Moor, O., Schwartzbach, M.I. (eds.) CC 2009. LNCS, vol. 5501, pp. 1–1. Springer, Heidelberg (2009)
24. Tian, C., Nagarajan, V., Gupta, R., Tallam, S.: Automated dynamic detection of busy-wait synchronizations. Softw., Pract. Exper. 39(11), 947–972 (2009)
25. Tlili, S., Debbabi, M.: Interprocedural and flow-sensitive type analysis for memory and type safety of c code. J. Autom. Reasoning 42(2-4), 265–300 (2009)

26. Wang, J., Ma, X., Dong, W., Xu, H.-F., Liu, W.: Demand-driven memory leak detection based on flow- and context-sensitive pointer analysis. J. Comput. Sci. Technol. 24(2), 347–356 (2009)
27. Wang, Y., Kelly, T., Kudlur, M., Lafortune, S., Mahlke, S.A.: Gadara: Dynamic deadlock avoidance for multithreaded programs. In: Draves, R., van Renesse, R. (eds.) OSDI, pp. 281–294. USENIX Association (2008)
28. Xiao, X., Lee, J.J.: A true o(1) parallel deadlock detection algorithm for single-unit resource systems and its hardware implementation. IEEE Trans. Parallel Distrib. Syst. 21(1), 4–19 (2010)
29. Xu, C., Che, Y., Fang, J., Wang, Z.: Optimizing adaptive synchronization in parallel simulators for large-scale parallel systems and applications. In: CIT, pp. 131–138. IEEE Computer Society, Los Alamitos (2010)
30. Yu, H., Xue, J., Huo, W., Feng, X., Zhang, Z.: Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In: Moshovos, A., Steffan, J.G., Hazelwood, K.M., Kaeli, D.R. (eds.) CGO, pp. 218–229. ACM, New York (2010)