Pointer Analysis for Structured Parallel Programs

RADU RUGINA
Cornell University
and
MARTIN C. RINARD
Massachusetts Institute of Technology

This paper presents a novel interprocedural, flow-sensitive, and context-sensitive pointer analysis algorithm for multithreaded programs that may concurrently update shared pointers. The algorithm is designed to handle programs with structured parallel constructs, including fork-join constructs, parallel loops, and conditionally spawned threads. For each pointer and each program point, the algorithm computes a conservative approximation of the memory locations to which that pointer may point. The algorithm correctly handles a wide range of programming language constructs, including recursive functions, recursively generated parallelism, function pointers, structures, arrays, nested structures and arrays, pointer arithmetic, casts between different pointer types, heap and stack allocated memory, shared global variables, and thread-private global variables. We have implemented the algorithm in the SUIF compiler system and used the implementation to analyze a set of multithreaded programs written in the Cilk programming language. Our experimental results show that the analysis has good precision and converges quickly for our set of Cilk programs.

Categories and Subject Descriptors: F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*

General Terms: Analysis, Languages

Additional Key Words and Phrases: Pointer analysis

1. INTRODUCTION

The use of multiple threads of control is quickly becoming a mainstream programming practice. Programmers use multiple threads for many reasons—

An earlier version of this paper appeared at the 1999 Conference on Programming Language Design and Implementation, Atlanta, Georgia, May 1999.
Authors' addresses: Radu Rugina, Cornell University, Computer Science Department, 4141 Upson Hall, Ithaca, NY 14853; email: rugina@cs.cornell.edu; Martin C. Rinard, MIT Laboratory for Computer Science, 545 Technology Square, NE43-620A, Cambridge, MA 02139; email: rinard@lcs.mit.edu.

to increase the performance of programs such as high-performance web servers that execute on multiprocessors, to hide the latency of events such as fetching remote data, for parallel programming on commodity SMPs, to build sophisticated user interface systems [Reppy 1992], and as a general structuring mechanism for large software systems [Hauser et al. 1993].

But multithreaded programs present a challenging problem for a compiler or program analysis system: the interactions between multiple threads make it difficult to extend traditional program analysis techniques developed for sequential programs to multithreaded programs.

One of the most important program analyses is pointer analysis, which extracts information about the memory locations to which pointers may point. Potential applications of pointer analysis for multithreaded programs include: the development of sophisticated software engineering tools such as race detectors and program slicers; memory system optimizations such as prefetching and moving computation to remote data; automatic batching of long latency file system operations; and to provide information required to apply traditional compiler optimizations such as constant propagation, common subexpression elimination, register allocation, code motion and induction variable elimination to multithreaded programs.

The difficulty of applying sequential analyses and optimizations to multithreaded programs is well known [Midkiff and Padua 1990]. A straightforward adaptation of these techniques would analyze all the possible interleavings of statements from the parallel threads [Cousot and Cousot 1984; Chow and Harrison 1992a]. However, this approach is not practical because of the combinatorial explosion in the number of potential program paths. Another adaptation of existing sequential analyses is to use flow-insensitive techniques [Andersen 1994; Steensgaard 1996] for the analysis of multithreaded programs [Hicks 1993; Zhu and Hendren 1997; Ruf 2000]. Such analyses ignore the flow of control in the program; they represent the program as a set of statements which can execute multiple times, in any possible order. Hence they automatically model all the interleavings of statements from the parallel threads. However, flow-insensitive analyses are known to be less precise than their flow-sensitive counterparts [Ryder et al. 2001]. The focus of this paper is the efficient, flow-sensitive analysis of pointers in multithreaded programs.

The key difficulty associated with the flow-sensitive pointer analysis for multithreaded programs is the potential for *interference* between parallel threads. Two threads interfere when one thread writes a pointer variable that another thread accesses (i.e. reads or writes). Interference between threads increases the set of locations to which pointers may point. Any pointer analysis algorithm for multithreaded programs must therefore characterize this interference if it is to generate correct information.

This paper presents a new interprocedural, flow-sensitive, and context-sensitive pointer analysis algorithm for multithreaded programs. The algorithm is designed to analyze programs with structured parallel constructs, including fork-join constructs, parallel loops, and conditionally spawned threads; the algorithm ignores synchronization constructs such as semaphores, locks, and critical sections, thus being conservative for programs that use these constructs. For each program point, our algorithm generates a *points-to graph* that specifies, for each pointer, the set of locations to which that pointer may point. To compute this graph in the presence of multiple threads, the algorithm extracts *interference information* in the form of another points-to graph that captures the effect of pointer assignments performed by the parallel threads. The analysis is adjusted to take this additional interference information into account when computing the effect of each statement on the points-to graph for the next program point.

We have developed a precise specification, in the form of dataflow equations, of the interference information and its effect on the analysis, and proved that these dataflow equations correctly specify a conservative approximation of the actual points-to graph.[1] We have also developed an efficient fixed-point algorithm that runs in polynomial time and solves these dataflow equations.

We have implemented this algorithm in the SUIF compiler infrastructure and used the system to analyze programs written in Cilk, a multithreaded extension of C [Frigo et al. 1998]. The implemented algorithm handles a wide range of constructs in multithreaded programs, including recursive functions and recursively generated parallelism, function pointers, structures and arrays, pointer arithmetic, casts between different pointer types, heap allocated memory, stack allocated memory, shared global variables, and thread-private global variables. Our experimental results show that the analysis has good overall precision and converges quickly.

This paper makes the following contributions:

—**Algorithm:** It presents a novel flow-sensitive, context-sensitive, interprocedural pointer analysis algorithm for multithreaded programs with structured parallelism. This algorithm is, to our knowledge, the first flow-sensitive pointer analysis algorithm for multithreaded programs.

—**Theoretical Properties:** It presents several important properties of the algorithm that characterize its correctness and efficiency. In particular, it shows that the algorithm computes a conservative approximation of the result obtained by analyzing all possible interleavings, and that the intraprocedural analysis of parallel threads runs in polynomial time with respect to the size of the program.

—**Experimental Results:** It presents experimental results for a sizable set of multithreaded programs written in Cilk. These results show that the analysis has good overall precision and converges quickly for our set of Cilk programs.

The rest of the paper is organized as follows. Section 2 presents an example that illustrates the additional complexity that multithreading can cause for pointer analysis. Section 3 presents the analysis framework and algorithm, and Section 4 presents the experimental results. Section 5 discusses some potential uses of the information that pointer analysis provides. We present related work in Section 6, and conclude in Section 7.

[1]Conservative in the sense that the points-to graph generated by our algorithm includes the points-to graph generated by first using the standard flow-sensitive algorithm for serial programs to analyze all possible interleaved executions, and then merging the results.

```
int x, y, z;
int *p;

main() {
1:  x = y = z = 0;
2:  p = &x;
3:  par {
4:    { *p = 1; }
5:    { p = &y;
6:      p = &z; }
7:  }
8:  *p = 2;
}
```

Fig. 1. Example multithreaded program.

2. EXAMPLE

Consider the simple multithreaded program in Figure 1. This program is written in a conceptual multithreaded language which uses a generic parallel construct `par`[2] to express the creation and execution of concurrent threads. This construct, also known as *fork-join* or *cobegin-coend*, is a basic parallel construct in many multithreaded languages.

The program manipulates a shared pointer `p`, and, at lines `4` and `8`, writes the memory locations to which `p` points. To have information about the actual memory locations that the indirect updates `*p = 1` and `*p = 2` write, further analyses and optimizations of this program need to know where `p` points to before these updates.

2.1 Parallel Execution

The execution of the program proceeds as follows. It first initializes all the variables and makes `p` point to `x`. Then, at line `3`, the program uses the `par` construct to create two parallel threads which concurrently access the shared pointer `p`. The first thread executes the statement at line `4`, which writes the value `1` into the memory location to which `p` points. The second thread executes the statements at lines `5` and `6`, which update `p` and make it point first to `y`, then to `z`. Any interleaved execution of the statements from these parallel threads may occur when the program runs, and all of these statements must complete before the execution proceeds beyond the `par` construct. There are three possible orders of execution of the statements from the parallel threads:

(1) The statement `*p = 1` executes first, before the statements `p = &y` and `p = &z`. In this case `p` still points to `x` when the statement `*p = 1` executes.
(2) The statement `*p = 1` executes second, after `p = &y` and before `p = &z`. Then `p` points to `y` when `*p = 1` executes.

[2]The parallel construct `par` is not part of the Cilk multithreaded language. This construct is expressed in Cilk by several `spawn` statements, each of which starts the execution of a new thread, and a `sync` statement, which blocks the parent thread until all spawned threads have completed.

(3) The statement *p = 1 executes last, after the statements p = &y and p = &z. In this case p points to z when the statement *p = 1 executes.

Any further analysis or optimization of this program must take all of these possibilities into account. The compiler must determine that the statement *p = 1 at line 4 assigns the value 1 to either x, y, or z, depending on the particular execution of the parallel program.

When both threads complete, execution proceeds beyond the end of the par construct at line 7. At this point, all the writes to the shared pointer p have completed and the last statement that writes p is p = &z at line 6. Therefore, p definitely points to z after the par construct and the statement *p = 2 will always write z, regardless of the particular execution of the parallel program.

This example illustrates how interference between pointer assignments and uses in parallel threads can affect the memory locations that the threads access. In particular, interference increases the set of memory locations to which pointers may point, which in turn increases the set of memory locations that pointer dereferences may access. Any sound pointer analysis algorithm must conservatively characterize the effect of this interference on the points-to relation.

2.2 Flow-Sensitive Analysis

To precisely characterize the memory locations to which p points, the compiler must take into account the flow of control within each thread and perform a *flow-sensitive* pointer analysis. In the example, a flow-sensitive analysis can precisely determine that the statement p = &z executes after the statement p = &y in all the possible executions of the parallel program. Such an analysis can therefore detect that p definitely points to z when the par statement completes and can conclude that the statement *p = 2 updates z in all executions of the program.

On the other hand, a flow-insensitive analysis would not take into account the order of execution of the statements p = &y and p = &z. Such an analysis would be too conservative and would determine that p points to either y or z after the par statement. It would therefore generate a spurious points-to edge from p to y, which does not occur in any execution of the program.

This example shows that the application of existing flow-insensitive pointer analyses to model the interleaving of statements from the parallel threads is not precise and may generate spurious points-to edges because it ignores the flow of control within threads. A precise, flow-sensitive multithreaded pointer analysis must characterize all the possible interleavings of statements from the parallel threads, but at the same time must take into account the flow of control within each thread.

3. THE ALGORITHM

This section presents the flow-sensitive pointer analysis algorithm for multithreaded programs. It first describes the basic framework for the analysis, then presents the dataflow equations for basic pointer assignment statements and `par` constructs. Finally, it describes the extensions for private global variables, interprocedural analysis, recursive functions, and function pointers.

3.1 Points-To Graphs and Location Sets

The algorithm represents the points-to relation using a *points-to graph* [Emami et al. 1994]. This representation is similar to the storage shape graph [Chase et al. 1990] and to the static shape graph [Sagiv et al. 1998] used in shape analysis algorithms. Each node in the graph represents a set of memory locations; there is a directed edge from one node to another node if one of the memory locations represented by the first node may point to one of the memory locations represented by the second node. Each node in the graph is implemented with a *location set* [Wilson and Lam 1995], which is a triple of the form $\langle$*name, offset, stride*$\rangle$ consisting of a variable name that describes a memory block, an offset within that block and a stride that characterizes the recurring structure of data vectors. A location set $\langle n, o, s\rangle$ corresponds to all sets of locations $\{o + is \mid i \in \mathbf{N}\}$ within block n. The location set for a scalar variable `v` is therefore $\langle \texttt{v}, 0, 0\rangle$; for a field `f` in a structure `s` is $\langle \texttt{s}, \texttt{f}, 0\rangle$ (here `f` is the offset of the field within the structure); for a set of array elements `a[i]` is $\langle \texttt{a}, 0, \texttt{s}\rangle$ (here `s` is the size of an element of `a`); and for a set of fields `a[i].f` is $\langle \texttt{a}, \texttt{f}, \texttt{s}\rangle$ (here `f` is the offset of `f` within the structure and `s` is the size of an element of `a`). Each dynamic memory allocation site has its own name, so the location set that represents a field `f` in a structure dynamically allocated at site `s` is $\langle \texttt{s}, \texttt{f}, 0\rangle$. Finally, all the elements of overlapping structures (such as the unions in C) are merged into a single location set. Therefore, if location sets have different memory block names, they represent disjoint sets of memory locations. This analysis, like other pointer analyses, assumes that programs do not violate the array bounds and that there are no assignments from integers to pointer variables.

Our analysis uses a special location set `unk`, the *unknown location set*, which characterizes the undefined initial pointer values and NULL pointers. The analysis treats the unknown location specially: it assumes that it is never dereferenced, which corresponds to a normal execution of the program. Dereferencing uninitialized or NULL pointers in C results in undefined execution behavior; the program continues its normal execution beyond a pointer dereference only if the dereferenced pointer is properly initialized and does not refer to a memory location represented by `unk`. Since our analysis assumes a normal execution of the program, in the case of store pointer assignments via potentially uninitialized or NULL pointers, the analysis updates only the explicit targets and leaves all the other pointers in the program unchanged. Similarly, in the case of load assignments via pointers that potentially point to the unknown location, the analysis assigns only the explicit, known targets of the dereferenced pointer to the variable in the left hand side of the assignment. This is the standard way of handling uninitialized variables in pointer analysis [Andersen 1994; Emami et al. 1994; Steensgaard 1996]. The use of the unknown location set also allows us to easily differentiate between *definite* pointers and *possible* pointers. If there is only one edge $\texttt{x} \rightarrow \texttt{y}$ from the location set `x`, `x` definitely points to `y`.

`x = &y`	Address-of Assignment
`x = y`	Copy Assignment
`x = *y`	Load Assignment
`*x = y`	Store Assignment

Fig. 2. Kinds of basic pointer assignment statements.

If `x` may point to `y` or it may be uninitialized, there are two edges from `x`: `x` $\to$ `y` and `x` $\to$ `unk`.

Each program defines a set L of location sets. We represent points-to graphs as sets of edges $C \subseteq L \times L$, and order points-to graphs using the set inclusion relation. This partial order defines a lattice whose meet operation is the set union $\cup$.

3.2 Basic Pointer Assignment Statements

To simplify the presentation of the algorithm, we assume that the program is preprocessed to a standard form where each pointer assignment statement is of one of the four kinds of *basic pointer assignment statements* in Figure 2.

More complicated pointer assignments can be reduced to a sequence of basic statements. In these basic assignment forms, `x` and `y` represent location sets. Hence an actual assignment like `s.f = v[i]` of the element `i` of the array `v` to the field `f` of the structure `s` is considered to be a basic assignment of the form `x = y`, since both `s.f` and `v[i]` can be expressed as location sets.

Our dataflow analysis algorithm computes a points-to graph for each program point in the current thread. It uses an *interference graph* to characterize the interference between the current thread and other parallel threads. The interference graph represents the set of edges that the other threads may create; the analysis of the current thread takes place in the context of these edges. As part of the analysis of the current thread, the algorithm extracts the set of edges that the thread creates. This set will be used to compute the interference graph for the analysis of other threads that may execute in parallel with the current thread.

Definition 3.2.1. Let L be the set of location sets in the program and $P = 2^{L\times L}$ the set of all points-to graphs. The *multithreaded points-to information MTI(p)* at a program point *p* of the parallel program is a triple $\langle C, I, E\rangle \in P^3$ consisting of:

—the *current points-to graph* C,
—the set I of *interference edges* created by all the other concurrent threads,
—the set E of *edges created by the current thread*.

We use dataflow equations to define the analysis of the basic statements. These equations use the *strong* flag which specifies whether the analysis of the basic statement performs a strong or weak update. Strong updates kill the existing edges of the location set being written, while weak updates leave the existing edges in place. Strong updates are performed if the analysis can identify a single pointer variable that is being written. If the analysis cannot

$$[\![st]\!]\langle C, I, E\rangle = \langle C', I', E'\rangle, \textit{where:}$$

$$C' = \begin{cases} (C - \textit{kill}) \cup \textit{gen} \cup I & \text{if } \textit{strong} \\ C \cup \textit{gen} \cup I & \text{if not } \textit{strong} \end{cases}$$
$$I' = I$$
$$E' = E \cup \textit{gen}$$

Fig. 3. Dataflow equations for basic statements.

identify such a variable (this happens if there are multiple location sets that denote the updated location or if one of the location sets represents more than one memory location), the analysis performs a weak update. Weak updates happen if there is uncertainty caused by merges in the flow of control, if the statement writes an element of an array, or if the statement writes a pointer in heap allocated memory.

If we denote by *Stat* the set of program statements, then the dataflow analysis framework is defined, in part, by a functional $[\![\ ]\!] : \textit{Stat} \to (P^3 \to P^3)$ that associates a transfer function $f \in P^3 \to P^3$ to every statement *st* in the program. As Figure 3 illustrates, we define this functional for basic statements using the *strong* flag and the *kill/gen* sets. Basic statements do not modify the interference information; the created edges are added to the set E of edges created by the current thread and to the current points-to graph C, and edges are killed in C only if the the *strong* flag indicates a strong update. The equations maintain the invariant that the interference information is contained in the current points-to graph.

Figure 4 defines the *gen* and *kill* sets and the *strong* flag for basic statements; Figure 5 shows how existing edges in the points-to graph (solid lines) interact with the basic statements to generate new edges (dashed lines). The definitions use the following dereference function (here 2^L is the set of all subsets of the set of location sets L):

$$\text{deref} : 2^L \times P \to 2^L , \quad \text{deref}(S, C) = \{y \in L \mid \exists x \in S \,.\, (x, y) \in C\}$$

We define a partial order relation $\sqsubseteq$ over the multithreaded points-to information P^3 using the inclusion relation between sets of points-to edges:

$$\langle C_1, I_1, E_1\rangle \sqsubseteq \langle C_2, I_2, E_2\rangle \quad \text{iff}$$
$$C_2 \subseteq C_1 \quad \text{and} \quad I_2 \subseteq I_1 \quad \text{and} \quad E_2 \subseteq E_1$$

This partial order defines a lattice on P^3. The top element in the lattice is the triple of empty sets of edges $\top = \langle \emptyset, \emptyset, \emptyset \rangle$; the meet operation is the component-wise union of sets of points-to edges:

$$\langle C_1, I_1, E_1\rangle \sqcup \langle C_2, I_2, E_2\rangle = \langle C_1 \cup C_2, I_1 \cup I_2, E_1 \cup E_2\rangle$$

It is easy to verify that the transfer functions for basic statements are monotonic in this lattice. This completes the definition of a full dataflow analysis framework for sequential programs. Our analysis therefore handles arbitrary sequential control flow constructs, including unstructured constructs.

`x = &y`	$\mathit{kill} = \{\mathtt{x}\} \times \mathrm{deref}(\{\mathtt{x}\}, C)$ $\mathit{gen} = \{\mathtt{x}\} \times \{\mathtt{y}\}$ $\mathit{strong} = (\mathtt{x}.\mathrm{stride} = 0 \;\text{ and }\; \text{not heap}(\mathtt{x})$
`x = y`	$\mathit{kill} = \{\mathtt{x}\} \times \mathrm{deref}(\{\mathtt{x}\}, C)$ $\mathit{gen} = \{\mathtt{x}\} \times \mathrm{deref}(\{\mathtt{y}\}, C)$ $\mathit{strong} = (\mathtt{x}.\mathrm{stride} = 0 \;\text{ and }\; \text{not heap}(\mathtt{x}))$
`x = *y`	$\mathit{kill} = \{\mathtt{x}\} \times \mathrm{deref}(\{\mathtt{x}\}, C)$ $\mathit{gen} = \{\mathtt{x}\} \times \mathrm{deref}(\mathrm{deref}(\{\mathtt{y}\}, C), C)$ $\mathit{strong} = (\mathtt{x}.\mathrm{stride} = 0 \;\text{ and }\; \text{not heap}(\mathtt{x}))$
`*x = y`	$\mathit{kill} = \mathrm{deref}(\{\mathtt{x}\}, C) \times \mathrm{deref}(\mathrm{deref}(\{\mathtt{x}\}, C), C)$ $\mathit{gen} = \mathrm{deref}(\{\mathtt{x}\}, C) \times \mathrm{deref}(\{\mathtt{y}\}, C)$ $\qquad - \{\mathtt{unk}\} \times L$ $\mathit{strong} = (\mathrm{deref}(\{\mathtt{x}\}, C) = \{\mathtt{z}\} \;\text{ and }$ $\qquad \mathtt{z}.\mathrm{stride} = 0 \;\text{ and }\; \text{not heap}(\mathtt{z}))$

Fig. 4. Dataflow information for basic statements.

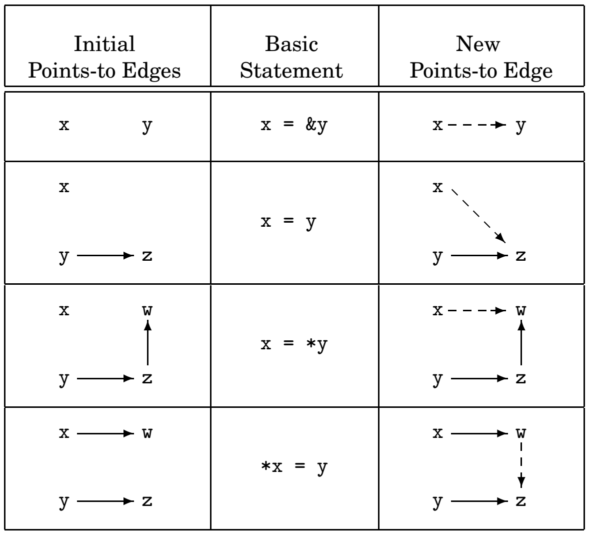

Fig. 5. New points-to edges for basic statements.

Finally, note that if the interference set of edges is empty, i.e., $I = \emptyset$, then the above algorithm reduces to the traditional flow-sensitive pointer analysis algorithm for sequential programs. Therefore, our algorithm for multithreaded programs may be viewed as a generalization of the algorithm for sequential programs.

$$[\![\, \mathrm{par}\{\{t_1\} \ldots \{t_n\}\} \,]\!] \, \langle C, I, E \rangle = \langle C', I, E' \rangle, \text{ where}$$

$$C' = \bigcap_{1 \le i \le n} C'_i$$

$$E' = E \cup \bigcup_{1 \le i \le n} E_i$$

$$C_i = C \cup \bigcup_{1 \le j \le n, j \ne i} E_j \qquad (\text{for } 1 \le i \le n)$$

$$I_i = I \cup \bigcup_{1 \le j \le n, j \ne i} E_j \qquad (\text{for } 1 \le i \le n)$$

$$[\![\, t_i \,]\!] \, \langle C_i, I_i, \emptyset \rangle = \langle C'_i, I_i, E_i \rangle \qquad (\text{for } 1 \le i \le n)$$

Fig. 6. Dataflow equations for par constructs.

3.3 Parallel Constructs

The basic parallel construct is the par construct (also referred to as fork-join or cobegin-coend), in which a *parent thread* starts the execution of several concurrent *child threads* at the beginning of the par construct, then waits at the end of the construct until all the child threads complete. It can then execute the subsequent statement.

We represent multithreaded programs using a parallel flow graph $\langle V, E \rangle$. Like a standard flow graph for a sequential program, the vertices of the graph represent the statements of the program, while the edges represent the potential flow of control between vertices. There are also *parbegin* and *parend* vertices. These come in corresponding pairs and represent, respectively, the beginning and end of the execution of a par construct. The analysis uses *begin* and *end* vertices to mark the begin and end of threads. There is an edge from each parbegin vertex to the begin vertex of each of its parallel threads, and an edge from the end vertex of each of its threads to the corresponding parend vertex.

We require there to be no edges between parallel threads, no edges from a vertex outside a par construct into one of its threads, and no edges from inside a thread to a vertex outside its par construct.

3.4 Dataflow Equations for Parallel Constructs

Figure 6 presents the dataflow equations for par constructs. The analysis starts with $\langle C, I, E \rangle$ flowing into the par construct and generates $\langle C', I, E' \rangle$ flowing out of the par construct. We next provide an intuitive explanation of these dataflow equations. Appendix B presents a formal proof of the soundness of these equations.

3.4.1 *Interference Information Flowing Into Threads.* We first consider the appropriate set of interference edges I_i for the analysis of the thread t_i. Because the thread may execute in parallel with any thread that executes in parallel with the par construct, all of the interference edges I that flow into the par statement should also flow into the thread. But the thread may also execute in parallel with all of the other threads of the par construct. So any local edges E_j created by the other threads should also be added into the interference information flowing into the analyzed thread. This reasoning determines the equation for the input interference edges flowing into thread t_i: $I_i = I \cup (\cup_{1 \leq j \leq n, j \neq i} E_j)$.

3.4.2 *Points-To Information Flowing Into Threads.* We next consider the current points-to graph C_i for the analysis of thread t_i. Because the thread may execute as soon as flow of control reaches the par construct, all of the edges in the current points-to graph C flowing into the par construct should clearly flow into the thread's input points-to graph C_i. But there are other edges that may be present when the thread starts its execution: specifically, any edges E_j created by the execution of the other threads in the par construct. These edges should also flow into the thread's input points-to graph C_i. This reasoning determines the equation for the current points-to graph that flows into thread t_i: $C_i = C \cup (\cup_{1 \leq j \leq n, j \neq i} E_j)$.

3.4.3 *Points-To Information Flowing Out of* par *Constructs.* We next consider the current points-to graph C' that flows out of the par construct. The analysis must combine the points-to graphs C'_i flowing out of the par construct's threads to generate C'. The points-to graph C'_i flowing out of thread t_i contains all of the edges created by the thread that are present at the end of its analysis, all of the edges ever created by the other parallel threads in the par construct, and all of the edges flowing into the par construct that were not killed by strong updates during the analysis of the thread.

We argue that intersection is the correct way to combine the points-to graphs C'_i flowing out of the threads to generate the final points-to graph C' flowing out of the par construct. There are two kinds of edges that should be in C': edges that are created by one of the par construct's threads and are still present at the end of that thread's analysis, and edges that flow into the par construct and are not killed by a strong update during the analysis of one of the threads. We first consider an edge created by a thread t_i and still present in the points-to graph C'_i flowing out of t_i. Because all of the points-to graphs C'_j flowing out of the other parallel threads contain all of the edges ever created during the analysis of t_i, the edge will be in the intersection.

We next consider any edge that flows into the par construct and is not killed by a strong update during the analysis of one of the threads. This edge will still be present at the end of the analysis of all of the threads, and so will be in the intersection.

Finally, consider an edge flowing into the par construct that is killed by a strong update during the analysis of one of the threads, say t_i. This edge will not be present in C'_i, and will therefore not be present in the intersection.

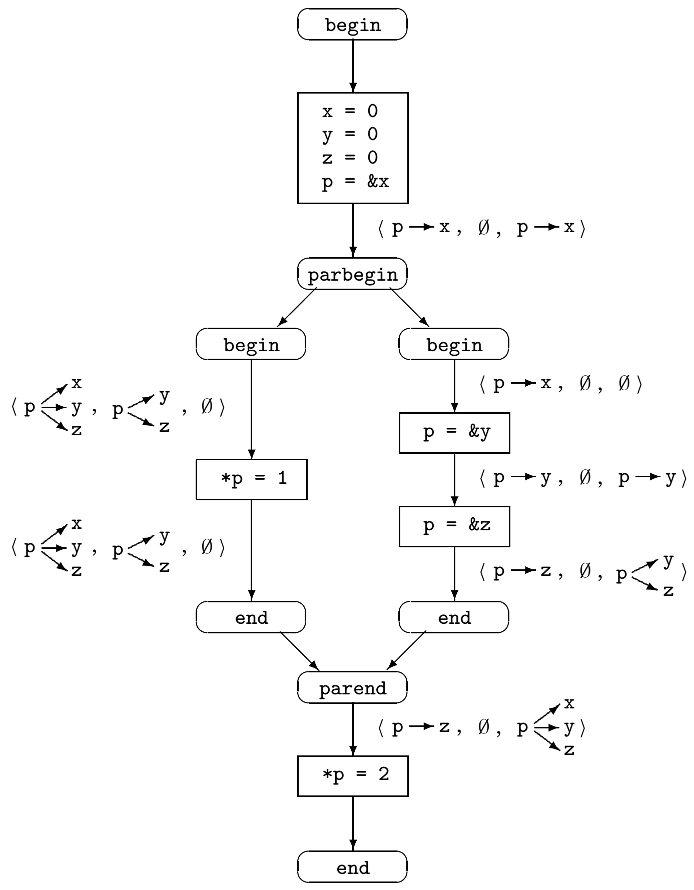

Fig. 7. Analysis result for example.

3.4.4 *Created Edges Flowing Out of* par *Constructs.* The set of edges E' flowing out of the par construct will be used to compute the interference information for threads that execute in parallel with the par construct. Since all of the par construct's threads can execute in parallel with any thread that can execute in parallel with the par construct, all of the edges E_j created during the analysis of the par construct's threads should be included in E'. This reasoning determines the equation $E' = E \cup (\cup_{1 \leq i \leq n} E_i)$.

3.5 Analysis of Example

Figure 7 presents the parallel flow graph for the example in Figure 1. Selected program points in the flow graph are annotated with the results of the analysis. Each analysis result is of the form $\langle C, I, E \rangle$, where C is the current points-to graph, I is the interference information, and E is the set of created edges.

Several points are worth mentioning. First, even though the edges p $\rightarrow$ y and p $\rightarrow$ z are not present in the points-to graph flowing into the par construct, they are present in the flow graph at the beginning of the first thread. This reflects the fact that the assignments p = &y or p = &z from the second thread may execute before any statement from the first thread. Second, even though the edges `p` $\rightarrow$ `y` and `p` $\rightarrow$ `z` are present in the flow graph at the end of the first thread, they are not present in the flow graph after the `par` construct. This reflects the fact the assignment `p = &z` from the second thread always executes after the assignments `p = &x` and `p = &y`. Therefore, after the `par` construct, `p` point to `z` instead of `x` or `y`.

3.6 Invariant on the Interference Information

We emphasize that the dataflow equations for basic statements from Figure 3 and the dataflow equations for `par` constructs from Figure 6 maintain the following invariant: $I \subseteq C$. This invariant means that the interference information is always included in the current points-to information, which is a key feature of our approach. Since the interference information contains all of the edges created by the concurrent threads, the invariant $I \subseteq C$ ensures that, at any program point, the current thread takes into account the effects of all the concurrent threads.

Proving that $I \subseteq C$ is true at any program point is relatively straightforward. In the case of basic statements, the relation $I' \subseteq C'$ holds after the execution of the statement *st* because the interference information is added to the current points-to information for both strong and weak updates. In the case of concurrent constructs, the proof is done by induction. We assume that $I \subseteq C$ before the `par` construct, and we prove that $I_i \subseteq C_i$ for each of the spawned threads t_i. This is directly obtained from the relations that give the points-to and interference information flowing into the thread t_i: $I_i = I \cup (\cup_{1 \leq j \leq n, j \neq i} E_j)$ and $C_i = C \cup (\cup_{1 \leq j \leq n, j \neq i} E_j)$.

3.7 The Fixed-Point Analysis Algorithm

The analysis algorithm uses a standard fixed-point approach to solve the dataflow equations in the sequential parts of code. There is a potential issue with analyzing `par` constructs. The dataflow equations for `par` constructs reflect the following circularity in the analysis problem: to perform the analysis for one of the `par` construct's threads, you need to know the analysis results from all of the other parallel threads. To perform the analysis on the other threads, you need the analysis results from the first thread.

The analysis algorithm breaks this circularity using a fixed-point algorithm. It first initializes the points-to information at each program point to $\langle \emptyset, \emptyset, \emptyset \rangle$. The exception is the program point before the entry node the overall graph, that is the point before the outermost `begin` node, where the points-to information is initialized to $\langle L \times \{\texttt{unk}\}, \emptyset, \emptyset \rangle$ so that each pointer is initially pointing to the unknown location. The algorithm then uses a standard worklist approach: every time a vertex's input information changes, the vertex is reanalyzed with the new information. Eventually the analysis converges on a fixed-point solution to the dataflow equations.

3.8 Complexity of Fixed-Point Algorithm

We derive the following complexity upper bound for the fixed-point intraprocedural analysis of parallel constructs. Let n be the number of statements in the program and l the number of abstract locations that the algorithm manipulates. The size of the points-to graphs is therefore $O(l^2)$ and so is the size of each points-to graph triple. Because the analysis computes a triple for each program point, the maximum size of all of the points-to graphs put together is $O(nl^2)$. If the fixed-point algorithm processes all the n nodes in the flow graph without adding an edge to one of the points-to graphs, it terminates. The algorithm therefore analyzes at most $O(n^2l^2)$ nodes. Finally, if all the pointer assignments in the program are basic statements, then at most two abstract locations appear in each statement. Hence $l \leq 2n$ and the complexity of the algorithm is $O(n^4)$. In practice, we expect that the complexity will be significantly smaller than this upper bound, and our experimental results support this expectation.

3.9 Precision of Analysis

We next discuss the precision of the multithreaded algorithm relative to an ideal algorithm that analyzes all interleavings of the statements from parallel threads, to which we refer as the *interleaved* algorithm. The basic idea of this algorithm is to eliminate parallel constructs from the flow graph using a product construction of the parallel threads, then use the standard algorithm for sequential programs to analyze the resulting interleaved sequential flow graph. The key result is that if there is no interference between the parallel threads, then the multithreaded algorithm yields the same results as the interleaved algorithm. We use here the standard definition of interference between parallel threads: two threads interfere if one thread writes a shared pointer that the other thread accesses.

Intuitively, this result comes from several observations. First, if threads do not interfere, they are independent and the algorithm can analyze each thread separately in isolation. In this case, the multithreaded analysis produces the same result as an algorithm that first performs a sequential analysis that ignores the edges created by other threads, and then merges these points-to edges at each program point. Second, if threads do not interfere, the interference points-to information for each thread precisely characterizes all the possible partial executions of the other concurrent threads: for each edge in the interference points-to information, there is a thread and a partial execution of the thread that creates that edge. Appendix C provides more details about the precision of the algorithm.

Hence, in the absence of interference, our analysis provides a safe, efficient way to obtain the precision of the ideal algorithm. Our experimental results show that, even for programs that concurrently access shared pointers, our multithreaded algorithm gives virtually the same results as the interleaved algorithm.

3.10 Parallel Loops

The parallel loop construct `parfor` executes, in parallel, a statically unbounded number of threads that execute the same loop body `parbody`. The analysis of a parallel loop construct is therefore equivalent to the analysis of a `par` construct that spawns an unknown number of concurrent threads executing the

$$[\![\,\texttt{parfor\{parbody\}}\,]\!]\,\langle C, I, E\rangle = \langle C_0', I, E \cup E_0\rangle, \text{ where}$$

$$[\![\ \texttt{parbody}\,]\!]\,\langle C \cup E_0, I \cup E_0, \emptyset\rangle = \langle C_0', I \cup E_0, E_0\rangle$$

Fig. 8. Dataflow equations for the parallel loop construct `parfor`.

same code. The system of dataflow equations for the parallel loop construct is therefore similar to the one for the `par` construct, but the number of equations in the new system depends on the dynamic number n of concurrent threads.

However, the identical structure of the threads allows simplifications in these dataflow equations. In particular, the analyses of the identical threads produce identical results. Hence the analyses will produce, for all threads, the same final points-to information, $C_i' = C_0', 1 \le i \le n$, and the same set of created edges, $E_i = E_0, 1 \le i \le n$. The interference points-to information is therefore the same for all of the threads: $I_i = I \cup \bigcup_{1 \le j \le n, j \ne i} E_j = I \cup E_0$. Here we conservatively assume that the loop creates at least two concurrent threads.

The dataflow equations for a parallel loop construct `parfor` are shown in Figure 8. They are derived from the equations from Figure 6 by imposing the requirement that all the spawned threads have the same final points-to information C_0' and the same set of created edges E_0. This produces identical dataflow equations for all the concurrent threads, therefore the analysis can solve a dataflow equation for a single thread (the second equation from Figure 8) to produce a result valid for all threads.

The approach of using a finite number of copies of the loop body for analyzing the interactions between a statically unbounded number of threads has also been used by other researchers [Krishnamurthy and Yelick 1996; Lee et al. 1998]. However, those specific analysis problems required two copies of the loop body, while in the context of pointer analysis our algorithm needs only one copy.

3.11 Conditionally Spawned Threads

The conditional parallel construct `parcond` is similar to the `par` construct. The statement `parcond`$\{\{t_1, c_1\}, \ldots, \{t_n, c_n\}\}$ *conditionally* starts the parallel execution of at most n child threads; it executes thread t_i only if its condition expression c_i is true. The program then waits at the end of the construct until all the child threads complete.

Figure 9 shows the dataflow equations for conditional parallel constructs. To characterize the fact that threads may not be executed if their condition evaluates to false, the analysis adjusts the way that the points-to graphs are combined at the synchronization point. Before taking the intersection of the points-to graphs from conditionally executed child threads, the analysis adds all of the edges killed during the analysis of each conditionally executed child thread back into the points-to graph flowing out of the child thread. This fact is captured by the equation $C' = \bigcap_{1 \le i \le n} (C_i' \cup C_i)$. This modification ensures that

$$[\![\, \texttt{parcond}\{\{t_1, c_1\} \ldots \{t_n, c_n\}\} \,]\!] \, \langle C, I, E \rangle = \langle C', I, E' \rangle, \text{ where}$$

$$C' = \bigcap_{1 \le i \le n} (C'_i \cup C_i)$$

$$E' = E \cup \bigcup_{1 \le i \le n} E_i$$

$$C_i = C \cup \bigcup_{1 \le j \le n, j \ne i} E_j \qquad (\text{for } 1 \le i \le n)$$

$$I_i = I \cup \bigcup_{1 \le j \le n, j \ne i} E_j \qquad (\text{for } 1 \le i \le n)$$

$$[\![\, t_i \,]\!] \, \langle C_i, I_i, \emptyset \rangle = \langle C'_i, I_i, E_i \rangle \qquad (\text{for } 1 \le i \le n)$$

Fig. 9. Dataflow equations for the conditional parallel construct `parcond`.

the analysis correctly reflects the possibility that the child thread may not be created and executed.

3.12 Private Global Variables

Multithreaded languages often support the concept of private global variables. Conceptually, each executing thread gets its own version of the variable, and can read and write its version without interference from other threads. Unlike shared variables, private global variables are not accessible from other threads. They are not part of the address space of other threads, so passing the address of a private variable to another thread results in a meaningless (unknown) address in the invoked thread.

Our analysis models the behavior of private global variables by saving and restoring points-to information at thread boundaries. When the analysis propagates the points-to information from a parbegin vertex to the begin vertex of one of its threads, it replaces the points-to information flowing into the begin vertex as follows. All private global variables are initialized to point to the unknown location, and all pointers to private global variables are reinitialized to point to the unknown location. At the corresponding parend vertex, the analysis processes the edges flowing out of the child threads to change occurrences of private global variables to the unknown location (these occurrences correspond to the versions from the child threads). The analysis then appropriately restores the points-to information for the versions of the private global variables of the parent thread.

3.13 Interprocedural Analysis

At each call site, the analysis must determine the effect of the invoked procedure on the points-to information. Our algorithm uses a generalization of the mapping/unmapping technique proposed by Emami, Ghiya and Hendren [Emami et al. 1994]. It maps the current points-to information into the name space of the invoked procedure, analyzes the procedure, then unmaps the result back into the name space of the caller. Like the analysis of Wilson and Lam [1995], our analysis caches the result every time it analyzes a procedure. Before it analyzes a procedure, it looks up the procedure and the analysis context in the cache to determine if it has previously analyzed the procedure in the same context. If so, it uses the results of the previous analysis instead of reanalyzing the procedure.

We generalize the notions of input contexts and partial transfer functions to multithreaded programs by incorporating the interference information and the set of edges created by procedures in their definitions. The input to a procedure includes the interference information and the result of the procedure includes the set of edges created by that procedure. The interference information is not part of the procedure result because the execution of the procedure doesn't change its interference information.

Definition 3.13.1. Given a procedure p, a *multithreaded input context* (MTC) is a pair $\langle C_p, I_p \rangle$ consisting of an input points-to graph C_p and a set of interference edges I_p. A *multithreaded partial transfer function* (MT-PTF) of p is a tuple $\langle C_p, I_p, C'_p, E'_p, r'_p \rangle$ that associates the input context $\langle C_p, I_p \rangle$ with an analysis result for p. This analysis result consists of an output points-to graph C'_p, a set E'_p of edges created by p, and a location set r'_p that represents the return value of the procedure.

As part of the mapping process, the analysis replaces all location sets that are not in the naming environment of the invoked procedure with anonymous placeholders called *ghost location sets*, which are similar to the *non-visible variables*, *invisible variables*, or *extended parameters* used in earlier interprocedural analyses [Landi and Ryder 1992; Emami et al. 1994; Wilson and Lam 1995]. Ghost location sets serve two purposes. First, they allow the algorithm to distinguish between the parameters and local variables of the current invocation of a recursive procedure from those of the previous invocations, thus increasing the precision of the analysis. Second, ghost location sets facilitate the caching of previous analysis results, thus reducing the number of generated contexts and improving the efficiency of the analysis. Although the analysis may build an exponential number of contexts in the worst case, the use of partial transfer functions and ghost location sets can significantly reduce this number in practice, to just one or two contexts per procedure [Wilson and Lam 1995].

Ghost locations are also similar to the *phantom nodes* or *outside nodes* from some existing escape analyses [Choi et al. 1999; Whaley and Rinard 1999; Salcianu and Rinard 2001]. However, because those analyses are compositional and analyze procedures with no information about the actual invocation contexts, the phantom and outside nodes are general placeholders that match nodes in any input context. Depending on the particular input context, the mapping process may merge together several nodes from caller procedure into a single placeholder, thus increasing the granularity of the objects and reducing the precision of the analysis. In contrast, in our algorithm, ghost location sets are created for known input contexts at call sites and each ghost location in our algorithm precisely models a single location set in the input context.

During the mapping process, the algorithm translates both the current points-to graph C and the interference information I from the name space of the caller into the name space of the invoked procedure. The mapping algorithm, which sets up the input context $\langle C, I \rangle$ for the analysis of the invoked procedure consists of the following four steps on graph C; for the interference graph I, the algorithm performs only the first three steps:

- —The local variables and formal parameters of the current procedure are represented using location sets. The algorithm maps these location sets to new ghost location sets.
- —The actual parameters of the procedure call are also represented using location sets. The algorithm assigns the actual parameters to the formal parameters of the invoked procedure.
- —The algorithm removes subgraphs that are not reachable from the global variables and formal parameters of the invoked procedure.
- —The algorithm adds the local variables from the invoked procedure to the current points-to information. It initializes all of the local variables to point to the unknown location set.

The algorithm next analyzes the invoked procedure in this analysis context to get a result $\langle C'_p, E'_p, r'_p \rangle$. It then unmaps this result back into the name space of the calling procedure. The unmapping process consists of the following steps on the current points-to graph C'_p; for the set of created edges E'_p and the set of locations where the return value r'_p points to, the algorithm performs only the first two steps:

- —The algorithm maps all of the formal parameters and local variables from the invoked procedure to the unknown location set.
- —The algorithm unmaps the ghost location sets back to the corresponding location sets that represent the local variables and formal parameters of the caller procedure.
- —The algorithm adds the unreachable subgraphs removed during the mapping process back into the current points-to graph.

The analysis continues after the procedure call with this context. Appendix A provides a formal definition of mapping and unmapping.

3.14 Recursive Procedures

As described so far, this algorithm does not terminate for recursive procedures. To eliminate this problem, we use a fixed-point approach similar to those for sequential context-sensitive analysis [Emami et al. 1994; Wilson and Lam 1995]. However, these algorithms use exactly two contexts for the analysis of each recursive procedure: a *recursive context* at the root of the recursion and an *approximate context* that merges all the other contexts that occur during the recursion. In contrast, our analysis does not impose a limit on the number of contexts in recursive computations; it generates new contexts as needed and reuses existing ones whenever the inputs match. This approach is more precise in general. Consider for example a recursive procedure that passes its arguments to the recursive calls, but swaps their order on even iterations.[3] In this case, our approach generates two contexts with mutually recursive calls. If the arguments of the recursive procedure initially point to distinct objects, our analysis can determine that the arguments still point to distinct objects in all recursive invocations. The use of an approximate context here would merge these contexts and the analysis would imprecisely report that the parameters may be aliased. Besides being more precise, we believe that handling recursive procedures using multiple possible contexts is also more natural. It is easier to implement, and usually generates few (at most two) contexts for each recursive procedure.

Our iterative algorithm works as follows. It maintains a current best analysis result for each context; each such result is initialized to $\langle \emptyset, \emptyset, \emptyset \rangle$. As the algorithm analyzes the program, the nested analyses of invoked procedures generate a stack of analysis contexts that models the call stack in the execution of the program. Whenever an analysis context is encountered for the second time on the stack, the algorithm does not reanalyze the procedure (this would lead to an infinite loop). It instead uses the current best analysis result and records the analyses that depend on this result. Whenever the algorithm finishes the analysis of a procedure and generates an analysis result, it merges the result into the current best result for the analysis context. If the current best result changes, the algorithm repeats all of the dependent analyses. The process continues until it terminates at a fixed point.

3.15 Linked Data Structures on the Stack

As described so far, the algorithm does not terminate for programs that use recursive procedures to build linked data structures of unbounded size on the call stack.[4] It would instead generate an unbounded number of analysis contexts as it unrolls the recursion. We eliminate this problem by recording the actual location sets from the program that correspond to each ghost location set. Whenever the algorithm encounters an analysis context with multiple ghost location sets that correspond to the same actual location set, it maps the ghost location sets to a single new ghost location set. This technique maintains the invariant that no analysis context ever has more ghost location sets than there are location sets in the program, which ensures that the algorithm generates a finite number of analysis contexts.

3.16 Function Pointers and Library Functions

The algorithm uses a case analysis to analyze programs with function pointers that may point to more than one function. At pointer-based call sites, the

[3]This pattern occurs in the Heat benchmark described in Section 4.

[4]The *pousse* benchmark described in Section 4, for example, uses recursion to build a linked list of unbounded size on the call stack. The nodes in the linked list are stack-allocated C structures; a pointer in each node points to a structure allocated in the stack frame of the caller.

algorithm analyzes all the possible callees, then merges the unmapped results at the end of each callee to obtain the points-to graph after the call statement. This algorithm is a straightforward generalization of an earlier algorithm used to analyze serial programs with this property [Emami et al. 1994].

The analysis also models the behavior of several standard library functions that manipulate or return pointers, including operations on strings, file handling, and memory allocation. The analysis assumes that all other library functions have no side effects on the pointer variables in the program and ignores such library calls.

3.17 Analysis of Cilk

Our target language, Cilk, provides two basic parallel constructs: the `spawn` and `sync` constructs. The `spawn` construct enables the programmer to create a thread that executes in parallel with the continuation of its parent thread. The parent thread can then use the `sync` construct to block until all outstanding threads spawned by the current thread complete. Returning from a procedure implicitly executes a `sync` and waits for all the threads spawned by that procedure to complete. Hence, spawned threads do not outlive the procedures that create them.

Our compiler recognizes structured uses of these constructs. A sequence of `spawn` constructs followed by a `sync` construct is recognized as a `par` construct; a `sync` construct preceded by a loop whose body is a `spawn` construct is recognized as a parallel loop; and a sequence of `spawn` constructs enclosed by if statements, followed by a `sync` construct is recognized as a `parcond` construct.

The difficulty is that these constructs may be used to create unstructured forms of parallelism that don't match any of the above structured constructs (i.e. `par`, `parfor`, and `parcond` constructs). Our analysis conservatively approximates the unstructured execution of parallel threads; it replaces the unstructured uses of `spawn` and `sync` statements with *approximate structured constructs*, as follows. The algorithm first determines the smallest subgraph G in the control flow graph that contains all the `spawn` statements that a particular `sync` may block for. Because spawned threads do not outlive the procedure that creates them, G is a subgraph of the intraprocedural control flow graph of the enclosing procedure. The analysis then creates a special thread t_G that executes the sequential computation in G, and skips the `spawn` statements. The analysis finally replaces the subgraph G with an approximate parallel construct that concurrently executes the thread t_G and the threads corresponding to all the `spawn` statements. If a spawned thread is part of a loop in G, the analysis treats it as a parallel loop that executes an unbounded number of copies of that thread. If a spawned thread is not executed on all paths to the `sync` statement, it is a conditionally spawned thread and the analysis uses the techniques presented in Section 3.11.

Intuitively, the correctness of this transformation comes from the fact that, in the approximate parallel construct, the spawned threads conservatively execute in parallel with all of the code in t_G, while in the original subgraph G, they execute concurrently only with subparts of the computation in t_G. Hence, this transformation provides a sound way to handle the unstructured uses of `spawn` and `sync` statements in Cilk programs.

3.18 Analysis Scope

The analysis is designed for multithreaded programs with structured parallelism. It can also handle unstructured forms of parallelism in the Cilk language using conservative, approximate structured constructs, as presented in Section 3.17. However, the algorithm does not handle general unstructured parallel constructs (such as POSIX threads or Java threads) or synchronization constructs (such as the `post` and `wait` constructs in parallel dialects of Fortran programs, rendezvous constructs in Ada, or the `wait` and `notify` constructs in Java).

There are several difficulties in these cases. Our current algorithm uses the structured form of concurrency to quickly find statements that can execute in parallel. Unstructured forms of concurrency and additional synchronization constructs can significantly complicate the determination of which statements can execute in parallel. Our current algorithm also exploits the fact that threads do not outlive their creating procedures to use a simple interprocedural abstraction with a single graph of created points-to edges to summarize the effect of each procedure on the analysis information. In effect, the analysis can treat the execution of the entire procedure call in the same way as other sequential statements, even though the procedure may create parallel threads internally. Supporting unstructured forms of concurrency in which threads may outlive their creating procedures would complicate the analysis and require the algorithm to use a more complicated abstraction. A related problem would be the need to match fork and join points across procedures and to propagate the information in the more complex abstraction across multiple procedure boundaries using mapping and unmapping.

4. EXPERIMENTAL RESULTS

We have implemented the multithreaded analysis in the SUIF compiler infrastructure. We built this implementation from scratch starting with the standard SUIF distribution, using no code from previous pointer analysis implementations for SUIF. We also modified the SUIF system to support Cilk, and used our implementation to analyze a sizable set of Cilk programs. Table I presents a list of the programs and several of their characteristics.

4.1 Benchmark Set

Our benchmark set includes all the programs in the Cilk distribution,[5] as well as several larger applications developed by researchers at MIT. The programs have been heavily optimized by hand to extract the maximum performance—in the case of *pousse*, for timed competition with other programs.[6] As a result, the

[5]Available at `http://supertech.lcs.mit.edu/cilk`.
[6]*Pousse* won the program contest associated with the ICFP '98 conference, and was undefeated in this contest.

Table I. Cilk Program Characteristics

Program	Lines of Code	Thread Creation Sites	Total(Pointer) Load Instructions	Total(Pointer) Store Instructions	Total(Pointer) Location Sets
barnes	1149	5	352 (318)	161(125)	1289(395)
block	342	19	140 (140)	9 (9)	546 (64)
cholesky	932	27	136 (134)	29 (29)	863(100)
cilksort	499	11	28 (28)	14 (14)	569 (65)
ck	505	2	85 (64)	49 (38)	571 (36)
fft	3255	48	461 (461)	335(335)	1883(103)
fib	53	3	1 (1)	0 (0)	451 (17)
game	195	2	9 (8)	9 (8)	508 (38)
heat	360	6	36 (36)	12 (12)	632 (34)
knapsack	122	3	14 (14)	6 (6)	444 (21)
knary	114	3	4 (4)	0 (0)	473 (20)
lu	594	24	16 (16)	13 (13)	688 (87)
magic	965	4	83 (83)	74 (74)	739(108)
mol	4478	33	1880(1448)	595(387)	2324(223)
notemp	341	17	136 (136)	6 (6)	546 (64)
pousse	1379	8	181 (161)	127(118)	905 (88)
queens	106	2	8 (8)	3 (3)	472 (23)
space	458	23	272 (272)	13 (13)	561 (70)

programs heavily use low-level C features such as pointer arithmetic and casts. Our analysis handles all of these low-level features correctly.

Most of the programs use divide and conquer algorithms. This approach leads to programs with recursively generated concurrency; the parameters of the recursive functions typically use pointers into heap or stack allocated data structures to identify the subproblem to be solved. The parallel sections in many of these programs manipulate sophisticated pointer-based data structures such as octrees, efficient sparse-matrix representations, parallel hash tables, and pointer-based representations of biological molecules.

We next discuss the application data in Table I. The Thread Creation Sites column presents the number of thread creation sites in the program; typically there would be several thread creation sites in a par construct or one thread creation site in a parallel loop. The Load and Store Instructions columns present, respectively, the number of load and store instructions in the SUIF representation of the program. The total number of load or store instructions appears first; the number in parenthesis is the number of instructions that access the value by dereferencing a pointer. Note that SUIF generates load or store instructions only for array accesses and accesses via pointers.

The Location Sets column presents the number of location sets in the program. The total number of location sets appears first; the number in parenthesis is the number of location sets that represent pointer values. These location set numbers include location sets that represent the formal parameters, local variables, and global variables. They do not include any ghost location sets generated as part of the analysis. The number of pointer location sets gives some idea of how heavily the programs use pointers. Bear in mind that the location set numbers include variables defined in standard include files; this is why a small application like *fib* has 451 total location sets and 17 pointer location sets. These numbers should be viewed as a baseline from which to calculate the number of additional location sets defined by the Cilk program.

4.2 Precision Measurements

We measure the precision of each analysis by computing, for each load or store instruction that dereferences a pointer, the number of location sets that represent the memory locations that the instruction may access. The fewer the number of target location sets per instruction, the more precise the analysis. Although a good metric for the precision of the points-to analysis depends on the specific application of the analysis information, the standard way to evaluate the precision of points-to analyses is to count the number of target locations for dereferenced pointers [Hind 2001].

We distinguish between *potentially uninitialized* pointers, or pointers with the unknown location set `unk` in the set of locations sets that represent the values to which the pointer may point, and *definitely initialized* pointers, or pointers without the unknown location set `unk`.

The precision measurements are complicated by the fact that each procedure may be analyzed for multiple points-to contexts. In different contexts, the load or store instructions may require different numbers of location sets to represent the accessed memory location. In this section we give two kinds of results, depending on the way we count the points-to sets for instructions in different contexts:

—*separate contexts:* loads and stores in different contexts are counted separately.
—*merged contexts:* results from all contexts of each procedure are merged together.

It is important to realize that the appropriate way of counting the target points-to sets depends on the intended use of the pointer analysis information. For example, if the information is used to verify statically that parallel calls are independent, appropriate precision metrics use analysis results from only those analysis contexts that appear at the parallel call sites. All other analysis contexts are irrelevant, because they do not affect the success or failure of the analysis that verifies the independence of parallel calls. Hence, for this kind of application, the analysis results for separate contexts are more relevant. On the other hand, if the analysis is used to optimize the generated code, and the compiler does not generate multiple specialized versions of the procedures, the optimization must take into account results from all of the analysis contexts. The MIT RAW compiler, for example, uses our pointer analysis algorithm in an instruction scheduling system [Barua et al. 1999]. It uses the pointer analysis information to find instructions that access memory from different memory modules. The success of the optimization depends on the addresses of the potentially accessed memory locations from all of the contexts. In this case, the analysis results for merged contexts are more relevant.

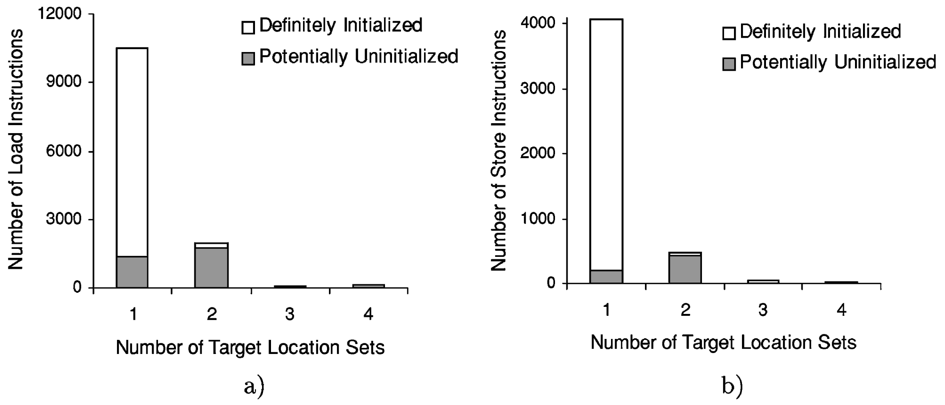

Fig. 10. Histogram of target points-to location sets in separate contexts for: a) load instructions and b) store instructions.

Our precision measurements count, for each load or store instruction, the number of location sets required to represent the accessed memory location. We present the precision measures using the histograms in Figure 10, which gives combined results for all the benchmarks for separate contexts. For each number of location sets, the corresponding histogram bar presents the number of load or store instructions that require exactly that number of location sets to represent the accessed memory location. Each bar is divided into a gray section and a white section. The gray section counts instructions whose dereferenced pointer is potentially uninitialized; the white section counts instructions whose dereferenced pointer is definitely initialized. Figure 10a presents the histogram for all contexts and all load instructions that use pointers to access memory in the Cilk programs. Figure 10b presents the corresponding histogram for store instructions.

These histograms show that the analysis gives good precision: for the entire set of programs, no instruction in any context requires more than four location sets to represent the accessed location. Furthermore, for the vast majority of the load and store instructions, the analysis is able to identify exactly one location set as the unique target, and that the dereferenced pointer is definitely initialized. According to the analysis, approximately one quarter of the instructions may dereference a potentially uninitialized pointer.

Table II breaks the counts down for each application for separate contexts. Each column is labeled with a number n; the data in the column is the sum over all procedures p and all analyzed contexts c of the number of load or store instructions in p that, in context c, required exactly n location sets to represent the accessed location. The number in parenthesis tells how many of the instructions dereference potentially uninitialized pointers. In *barnes*, for example, 715 load instructions in all the analyzed contexts require exactly one location set to represent the stored value; of these 715, only 79 used a potentially uninitialized pointer.

Table III presents similar results for merged contexts. Unlike the results from separate contexts, which count the location sets for each instruction

Table II. Per-Program Counts of the Number of Location Sets Required to Represent an Accessed Location—Separate Contexts, with Ghost Location Sets

	Load Instructions				Store Instructions			
	Number of Read Location Sets				Number of Written Location Sets			
Program	1	2	3	4	1	2	3	4
barnes	715 (79)	376(199)	2 (2)	147(147)	306 (7)	65 (39)	–	20 (20)
block	276 (2)	–	–	–	18 (0)	–	–	–
cholesky	94 (68)	624(624)	–	–	143 (20)	90 (90)	–	–
cilksort	40 (1)	–	–	–	19 (0)	–	–	–
ck	121 (3)	–	–	–	78 (0)	–	–	–
fft	1761 (1)	1 (0)	–	–	1287 (0)	–	–	–
fib	1 (1)	–	–	–	–	–	–	–
game	16 (0)	–	–	–	17 (0)	–	–	–
heat	103 (16)	33 (28)	5 (5)	–	21 (15)	30 (30)	–	–
knapsack	14 (0)	–	–	–	6 (0)	–	–	–
knary	4 (4)	–	–	–	–	–	–	–
lu	15 (0)	1 (1)	–	–	13 (0)	–	–	–
magic	114 (2)	–	–	–	82 (0)	–	–	–
mol	5549(1141)	871(871)	58(36)	–	1470(128)	265(265)	53(0)	–
notemp	136 (2)	–	–	–	7 (0)	–	–	–
pousse	765 (40)	14 (14)	3 (3)	–	578 (40)	–	–	–
queens	13 (3)	3 (3)	–	–	12 (0)	–	–	–
space	788 (2)	12 (0)	–	–	14 (0)	20 (0)	–	–

Table III. Per-Program Counts of the Number of Location Sets Required to Represent an Accessed Location—Merged Contexts, Ghost Location Sets Replaced By Corresponding Actual Location Sets

	Load Instructions					Store Instructions				
	Number of Read Location Sets					Number of Written Location Sets				
Program	1	2	3	4	12	1	2	3	4	6
barnes	146 (10)	114 (26)	2 (2)	53(47)	3 (0)	92 (2)	25 (6)	–	8(8)	–
block	136 (2)	4 (0)	–	–	–	–	9 (0)	–	–	–
cholesky	2 (0)	132(132)	–	–	–	7 (0)	22(22)	–	–	–
cilksort	16 (1)	12 (0)	–	–	–	9 (0)	5 (0)	–	–	–
ck	53 (3)	11 (0)	–	–	–	19 (0)	19 (0)	–	–	–
fft	144 (1)	–	–	317 (0)	–	18 (0)	–	–	317(0)	–
fib	1 (1)	–	–	–	–	–	–	–	–	–
game	5 (0)	3 (0)	–	–	–	4 (0)	4 (0)	–	–	–
heat	7 (2)	17 (0)	12(12)	–	–	2 (0)	5 (5)	5 (5)	–	–
knapsack	14 (0)	–	–	–	–	6 (0)	–	–	–	–
knary	4 (4)	–	–	–	–	–	–	–	–	–
lu	15 (0)	1 (1)	–	–	–	13 (0)	–	–	–	–
magic	67 (2)	16 (0)	–	–	–	61 (0)	13 (0)	–	–	–
mol	1113(163)	316(315)	19(12)	–	–	310(37)	61(58)	6 (0)	6(0)	4 (0)
notemp	136 (2)	–	–	–	–	5 (0)	1 (0)	–	–	–
pousse	101 (8)	56 (2)	1 (0)	3 (3)	–	64 (8)	44 (0)	10 (0)	–	–
queens	3 (3)	3 (1)	2 (2)	–	–	1 (0)	2 (0)	–	–	–
space	264 (2)	8 (0)	–	–	–	–	13 (0)	–	–	–

multiple times, once for each analysis context, the results in Table III merge all the analysis contexts and then count the location sets in the combined information, once for each instruction. There is also a difference in the way ghost location sets are counted. If a ghost location set represents the accessed memory location, Table II counts just the ghost location set, while Table III counts the actual location sets that were mapped to that ghost location set during the analysis.

We next discuss the sources that lead our analysis to report potentially initialized pointers. Arrays of pointers represent one such source. Because pointer analysis is not designed to recognize when every element of an array of pointers has been initialized, the analysis conservatively generates the result that each pointer in an array of pointers is potentially uninitialized. In general, it may be very difficult to remove this source of imprecision. For our set of measured Cilk programs, however, it would usually be straightforward to extend the analysis to recognize the loops that completely initialize the arrays of pointers.

NULL pointers are another source of potentially uninitialized pointers. In our analysis, NULL pointers are represented by the unknown location set. In some cases, NULL is passed as an actual parameter to a procedure containing conditionally executed code that dereferences the corresponding formal parameter. Although this code does not execute if the parameter is NULL, the algorithm does not perform the control-flow analysis required to realize this fact. The result is that, in contexts with the formal parameter bound to NULL, instructions in the conditionally executed code are counted as dereferencing potentially uninitialized pointers.

4.3 Comparison with Interleaved Analysis

Ideally, we would evaluate the precision of the algorithm presented in this paper, the Multithreaded algorithm, by comparing its results with those of the Interleaved algorithm, which uses the standard algorithm for sequential programs to analyze all possible interleavings. But the intractability of the Interleaved algorithm makes it impractical to use this algorithm even for comparison purposes.

We instead developed an extension of the standard flow-sensitive, context-sensitive algorithm for sequential programs, the Sequential algorithm, which ignores parbegin and parend vertices and analyzes the threads of the parbegin and parend in the order in which they appear in the program text. In effect, parallel threads are analyzed as if they execute sequentially; the algorithm analyzes only a particular interleaving and is therefore unsound for all the possible executions of the parallel program. However, it generates a less conservative result than the Interleaved algorithm and therefore provides an upper bound on the achievable precision.

We have then compared the results of the Sequential and Multithreaded algorithms. Since the Sequential and Multithreaded analysis algorithms generate analysis contexts that are incomparable because of the interference information, we have considered the results for merged contexts. For our set of benchmarks, the Sequential algorithm produced virtually identical location set counts as the Multithreaded analysis results shown in Table III. For *pousse*, the counts differ slightly because of the handling of private variables in the Multithreaded analysis. We conclude that, at least by this metric, the Sequential and Multithreaded analyses provide virtually identical

Table IV. Analysis Measurements

Program	Total Number of Parallel Construct Analyses	Mean Number of Iterations per Analysis	Mean Number of Threads per Analysis
barnes	12	2.00	2.00
block	13	1.00	3.85
cholesky	109	1.83	4.11
cilksort	8	1.00	2.50
ck	3	1.00	2.00
fft	182	1.73	3.50
fib	1	1.00	2.00
game	3	1.00	2.00
heat	8	1.62	2.00
knapsack	1	1.00	2.00
knary	1	1.00	2.00
lu	10	1.00	2.80
magic	24	1.00	2.00
mol	99	1.18	2.27
notemp	15	1.00	2.53
pousse	9	1.22	3.33
queens	8	2.25	2.00
space	15	1.00	6.80

precision. Recall that the Sequential analysis provides an upper bound on the precision attainable by the Interleaved algorithm. We therefore conclude that the Multithreaded algorithm and Interleaved algorithms provide virtually identical precision, at least for this metric and this set of benchmark programs.

4.4 Analysis Measurements

To give some idea of the complexity of the analysis in practice, we measured the total number of parallel construct analyses and the number of iterations required for each analysis to reach a fixed point. The use of other parallel constructs complicates the accounting. Here we count each synchronization point in the program as a parallel construct unless the synchronization point waits for only one thread. There is a synchronization point at the end of each `par` construct, each parallel loop `parfor`, each conditional parallel construct `parcond`, and each approximate structured construct (see Section 3.17). Table IV presents these numbers. This table includes the total number of analyses of parallel constructs, the mean number of iterations required to reach a fixed point for that parallel construct, and the mean number of threads analyzed each time. These numbers show that the algorithm converges quickly, with the mean number of iterations always less than or equal to 2.25. We note that it is possible for the analysis to converge in one iteration if the parallel threads create local pointers, use these pointers to access data, but never modify a pointer that is visible to another parallel thread. The mapping and unmapping process that takes place at procedure boundaries ensures that the algorithm correctly models this lack of external visibility.

Table V. Analysis Times (in Seconds) for Sequential and Multithreaded Analysis Algorithms

Program	Sequential Analysis Times	Multithreaded Analysis Times
barnes	48.45	49.75
blockedmul	0.78	0.93
cholesky	6.92	17.48
cilksort	0.32	0.36
ck	0.27	0.29
fib	0.01	0.01
fft	5.60	10.80
game	0.24	0.50
heat	0.40	0.82
knapsack	0.04	0.04
knary	0.02	0.03
lu	0.39	0.43
magic	1.77	6.26
mol	125.17	151.18
notempmul	0.53	0.73
pousse	5.34	7.77
queens	0.07	0.15
space	1.62	1.90

4.5 Analysis Times

Table V presents the analysis times for the Sequential and Multithreaded algorithms. These numbers should give a rough estimate of how much extra analysis time is required for multithreaded programs. Although the Multithreaded algorithm always takes longer, in most cases the analysis times are roughly equivalent. There are a few outliers such as *cholesky* and *fft*. In general, the differences in analysis times are closely correlated with the differences in the number of contexts generated during the analysis.

5. POTENTIAL USES

We foresee two primary uses for pointer analysis: to enable the optimization and transformation of multithreaded programs, and to build software engineering and program understanding tools.

5.1 Current Uses

To date, our pointer analysis algorithm has been used in several projects: an instruction scheduling project, a C-to-silicon parallelizing compiler, two bitwidth analysis projects, a project for software-managed caches, and a symbolic analysis project for divide and conquer programs.

The MIT RAWCC compiler uses our analysis to disambiguate the targets of load and store instructions [Barua et al. 1999]. The goal is to exploit instruction-level parallelism and to determine statically which memory modules may be accessed by specific instructions. The MIT DeepC compiler, which translates C code directly to silicon, uses our pointer analysis to split large memory banks into smaller memories with smaller address lengths

[Babb et al. 1999]. Each piece of combinational logic is then connected only to the accessed small memories, thus exploiting a finer-grain kind of parallelism than the RAWCC compiler. The DeepC compiler also uses our pointer analysis to perform bitwidth analysis, which determines the number of bits required to store values computed by the program [Stephenson et al. 2000]. Memory disambiguation provided via pointer analysis information enables more precise results than the conservative treatment of loads and stores. Our pointer analysis package was similarly used in the CMU PipeWrench project for improving precision of the bitwidth analysis [Budiu et al. 2000]. Another project, FlexCache, developed at MIT and at UMass Amherst, uses our analysis as part of an effort to deliver a software-only solution for managing on-chip data caches [Moritz et al. 2000]. The goal is to use a combination of compiler analyses, including pointer analysis, to reduce the number of software cache-tag lookups.

Finally, we have used the pointer analysis results as a foundation for the symbolic analysis of divide and conquer algorithms [Rugina and Rinard 1999, 2000]. The goal here is to use the extracted symbolic information to parallelize sequential programs, to check for data races in multithreaded programs, to eliminate redundant bounds checks in type-safe languages, or to check for array bounds violations in type-unsafe languages. For efficiency reasons, divide and conquer programs often access memory using pointers and pointer arithmetic. Our analysis algorithm provides the pointer analysis information required to symbolically analyze such pointer-intensive code.

In the symbolic analysis of divide and conquer programs, we use the multithreaded version of the algorithm for all the analyses except automatic parallelization. The other projects, as well as the parallelization of divide and conquer programs use the sequential version of the algorithm.

5.2 Software Engineering Uses

We believe that the software engineering uses will be especially important. Multithreaded programs are widely believed to be much more difficult to build and maintain than sequential programs. Much of the difficulty is caused by unanticipated interference between concurrent threads. In the worst case, this interference can cause the program to fail nondeterministically, making it very difficult for programmers to reproduce and eliminate bugs.

Understanding potential interactions between threads is the key to maintaining and modifying multithreaded programs. So far, the focus has been on dynamic tools that provide information about interferences in a single execution of the program [Savage et al. 1997; Cheng et al. 1998]. Problems with these tools include significant run-time overhead and results that are valid only for a single test run. Nevertheless, they provide valuable information that programmers find useful.

Accurate pointer analysis of multithreaded programs enables the construction of tools that provide information about all possible executions, not a single run. In fact, we have developed such a tool [Rugina and Rinard 2000]; it uses pointer analysis information, augmented by symbolic bounds information, to generate, for each read or write, the set of potentially accessed memory locations. It then correlates loads and stores from parallel threads to identify statements that could interfere and statically detect potential data races. Programmers can use this kind of information to understand the interactions between threads, identify all of the pieces of code that could modify a given variable, and rule out some hypothesized sources of errors. Such tools make it much easier to understand, modify and debug multithreaded programs.

5.3 Program Transformation Uses

Accesses via unresolved pointers prevent the compiler from applying standard optimizations such as constant propagation, code motion, register allocation and induction variable elimination. The basic problem is that all of these optimizations require precise information about which variables program statements access. But an access via an unresolved pointer could, in principle, access any variable or memory location. Unresolved pointer accesses therefore prevent the integrated optimization of the surrounding code. Pointer analysis is therefore required for the effective optimization of multithreaded programs.

It is also possible to use pointer analysis to optimize multithreaded programs that use graphics primitives, file operations, database calls, network or locking primitives, or remote memory accesses. Each such operation typically has overhead from sources such as context switching or communication latency. It is often possible to eliminate much of this overhead by batching sequences of operations into a single larger operation with the same functionality [Diniz and Rinard 1997; Bogle and Liskov 1994; Zhu and Hendren 1998]. Ideally, the compiler would perform these *batching transformations* automatically by moving operations to become adjacent, then combining adjacent operations. Information from pointer analysis is crucial to enabling the compiler to apply these transformations to code that uses pointers.

Finally, pointer analysis can be used in problems from distributed computing. Programs in such systems often distribute data over several machines. Information about how parts of the computation access data could be used to determine if data is available locally, and if not, whether it is better to move data to computation or computation to data [Carlisle and Rogers 1995]. Pointer analysis could also be used to help characterize how different regions of the program access data, enabling the application of consistency protocols optimized for that access pattern [Falsafi et al. 1994].

6. RELATED WORK

We discuss three areas of related work: pointer analysis for sequential programs, escape analysis for Java programs, and the analysis of multithreaded programs.

6.1 Pointer Analysis for Sequential Programs

Pointer analysis for sequential programs is a relatively mature field [Chase et al. 1990; Landi and Ryder 1992; Choi et al. 1993; Emami et al. 1994; Andersen

1994; Ruf 1995; Wilson and Lam 1995; Steensgaard 1996; Shapiro and Horwitz 1997; Diwan et al. 1998; Rountev and Chandra 2000; Das 2000; Heintze and Tardieu 2001]. A detailed survey of existing techniques in pointer analysis can be found in Hind [2001]. We classify analyses with respect to two properties: flow sensitivity and context sensitivity.

Flow-sensitive analyses take the statement ordering into account. They typically use dataflow analysis to perform an abstract interpretation of the program and produce a points-to graph or set of alias pairs for each program point [Landi and Ryder 1992; Choi et al. 1993; Emami et al. 1994; Ruf 1995; Wilson and Lam 1995].

Flow-insensitive analyses, as the name suggests, do not take statement ordering into account, and typically use some form of constraint-based analysis to produce a single points-to graph that is valid across an entire analysis unit [Andersen 1994; Steensgaard 1996; Shapiro and Horwitz 1997; O'Callahan and Jackson 1997; Das 2000; Heintze and Tardieu 2001]. The analysis unit is typically the entire program, although it is possible to use finer analysis units such as the computation rooted at a given call site. Researchers have proposed several flow-insensitive pointer analysis algorithms with different degrees of precision. In general, flow-sensitive analyses provide a more precise result than flow-insensitive analyses [Ryder et al. 2001], although it is unclear how important this difference is in practice. Finally, flow-insensitive analyses extend trivially from sequential programs to multithreaded programs. Because they are insensitive to the statement order, they trivially model all the interleavings of the parallel executions.

Roughly speaking, context-sensitive analyses produce an analysis result for each different calling context of each procedure. Context-insensitive analyses, on the other hand, produce a single analysis result for each procedure, typically by merging information from different call sites into a single analysis context. This approach may lose precision because of interactions between information from different contexts, or because information flows between call sites in a way that does not correspond to realizable call/return sequences. Context-sensitive versions of flow-sensitive analyses are generally considered to be more accurate but less efficient than corresponding context-insensitive versions, although it is not clear if either belief is true in practice [Ruf 1995; Wilson and Lam 1995].

A specific kind of imprecision in the analysis of recursive procedures makes many pointer analysis algorithms unsuitable for our purposes. We used our analysis as a foundation for race detection and symbolic array bounds checking of multithreaded programs with recursively generated concurrency. This application requires an analysis that is precise enough to recognize independent calls to recursive procedures, even when the procedures write data allocated on the stack frame of their caller. The only analyses that satisfy this requirement even for sequential programs are both flow sensitive and use some variant of the concept of invisible variables [Emami et al. 1994; Wilson and Lam 1995]. But even these analyses lose precision by limiting the number of contexts used to analyze the recursive computation. They approximate the contexts of all activations except the root of the recursion with a single analysis context. In contrast, our approach generates new contexts when needed and models the recursive computation with using mutual recursion between these contexts. This approach turned out to be critical for some of our applications.

6.2 Escape Analysis Algorithms for Java Programs

In the last few years, researchers have developed several escape analysis algorithms for multithreaded Java programs [Whaley and Rinard 1999; Blanchet 1999; Bogda and Hoelzle 1999; Choi et al. 1999; Ruf 2000]. All of these algorithms use some approximation of the reachability information between objects to compute the escape information. This reachability information is in turn an approximation of the points-to information between objects. All of the algorithms use the escape information for stack allocation and synchronization elimination. Because Java is a multithreaded language, these algorithms were designed to analyze multithreaded programs; they can all handle unstructured parallel constructs such as the Java threads. All of these algorithms, however, only analyze single threads, and are designed to find objects that are accessible to only the current thread. If an object escapes the current thread, either a reference to the object was written into a static class variable or because the object becomes accessible to another thread, it is marked as globally escaping, and there is no attempt to recapture the object by analyzing the interactions between the threads that access the object. These algorithms are therefore fundamentally sequential program analyses that have been adjusted to ensure that they operate conservatively in the presence of parallel threads.

Salcianu and Rinard [2001] have also developed, in research performed after the work presented in this paper, a new pointer and escape analysis that is able to analyze interactions between concurrent threads. Unlike previous escape analyses, this algorithm can detect, for instance, objects that are accessed by multiple threads but do not escape a certain multithreaded computation. However, the algorithm still performs weak updates (i.e. does not kill points-to edges) for the objects that are accessed by more than one thread. In contrast, the algorithm presented in this paper performs strong updates on shared pointers and can characterize more precisely the interactions between parallel threads.

6.3 Analysis of Multithreaded Programs

The analysis of multithreaded programs is an area of increasing recent interest [Rinard 2001]. It is clear that multithreading can significantly complicate program analysis [Midkiff and Padua 1990], but a full range of standard techniques has yet to emerge. In this section we discuss two main directions: extending traditional analyses for sequential programs to work with multithreaded programs, and analyses designed to enable optimizations or detect errors that are specific to multithreaded programs.

6.3.1 *Extending Analyses for Sequential Programs.* The straightforward approach for the flow-sensitive analysis of multithreaded programs is to analyze all possible interleavings of statements from the parallel threads, using a representation of the program consisting of configurations (states) and transitions between these configurations [Cousot and Cousot 1984; Chow and Harrison 1992a]. But this approach is not practical because of the combinatorial explosion in the number of states required to represent all the interleavings. To attack the state space explosion problem, researchers have proposed several ways of reducing the generated state space [Valmari 1990; Chow and Harrison 1992b; Godefroid and Wolper 1994]. These techniques rely on the fact that there is little interaction between the parallel threads; in the worst case they are not able to reduce the state space and for many programs the reduced space may be still unmanageable. Moreover, these techniques do not apply to pointer-based programs. In fact, the application of these techniques to programs with pointers would require points-to information.

Other work in the area of flow-sensitive multithreaded analysis extended the traditional sequential analyses and concepts for sequential programs to multithreadead programs. This included reaching definitions [Sarkar 1997], constant propagation [Knoop 1998; Lee et al. 1998], and code motion [Knoop and Steffen 1999] for multithreaded programs; concurrent static single assignment forms [Srinivasan et al. 1993; Lee et al. 1999]; and dataflow frameworks for bitvector problems [Knoop et al. 1996] or for multithreaded programs with copy-in, copy-out memory semantics [Grunwald and Srinivasan 1993]. Again, none of these frameworks and algorithms applies to pointer analysis and the application of these frameworks to programs with pointers would require pointer analysis information.

Other researchers have directly used existing flow-insensitive techniques to analyze multithreaded programs, in part because the reduced precision of these algorithms did not have a critical impact for their applications. Zhu and Hendren [1997] designed a set of communication optimizations for parallel programs and Ruf [2000] developed a technique for removing unnecessary synchronization in multithreaded Java programs; they both use flow-insensitive pointer analysis techniques [Steensgaard 1996] to detect pointer variable interference between parallel threads. Hicks [1993] also has developed a flow-insensitive analysis for a multithreaded language.

6.3.2 *Analyses and Verifications Specific to Multithreaded Programs.* Other related areas of research explore analyses, optimizations, and verifications that are specific to multithreaded programs. These include the analysis of synchronization constructs; deadlock detection; data race detection; communication and synchronization optimizations; and analyses for memory models. We next discuss each of these areas in turn.

The analysis of synchronization constructs is aimed at characterizing how the synchronization actions temporally separate the execution of various program fragments. The compiler can further use this information to precisely detect statements that may execute concurrently. Several such analyses have been developed to trace the control transfers associated with synchronization constructs such as the `post` and `wait` constructs in parallel dialects of Fortran [Callahan and Subhlok 1988; Emrath et al. 1989; Callahan et al. 1990], the Ada rendezvous constructs [Taylor 1983; Duesterwald and Soffa 1991; Masticola and Ryder 1993; Dwyer and Clarke 1994], and the `wait` and `notify` constructs in Java [Naumovich and Avrunin 1998; Naumovich et al. 1999]. But these are not designed to analyze the effects of accesses to shared pointers: they either assume the absence of shared pointers or they don't analyze data accesses at all. In contrast, our analysis is designed to analyze shared pointer accesses, not the synchronization structure of the program; our algorithm analyzes programs with structured parallelism and simple synchronization, where the detection of concurrent statements is straightforward from the program source. We therefore consider that synchronization analyses are orthogonal to our pointer analysis and we believe that our algorithm can be extended with such analysis techniques to handle more complex synchronization constructs.

A deadlock traditionally occurs from circular waiting to acquire resources, and is a classic problem in multithreaded computing. Deadlock detection in multithreaded programs is closely related to synchronization analysis. The compiler must analyze the synchronization structure of the program to detect if it may lead to deadlocks between concurrent threads. Researchers have developed a variety of analyses for detecting potential deadlocks in Ada programs that use rendezvous synchronization [Taylor 1983; Dillon 1990; Masticola and Ryder 1990; Long and Clarke 1991; Corbett 1996; Blieberger et al. 2000]. A rendezvous takes place between a call statement in one thread and an accept statement in another. The analyses match corresponding call and accept statements to determine if every call will eventually participate in a rendezvous. Again, these techniques focus on the analysis of the synchronization structure rather than the effects of shared data accesses.

A data race occurs when two parallel threads access the same memory location without synchronization and one of the accesses is a write. Because data races are almost always the result of programmer error, many researchers have developed tools designed to find and help eliminate data races. Several approaches have been developed to attack this problem, including static program specifications and verifications [Sterling 1994; Detlefs et al. 1998], augmented type systems to enforce the synchronized access to shared data [Flanagan and Abadi 1999; Flanagan and Freund 2000; Boyapati and Rinard 2001], and dynamic detection of data races based on program instrumentation [Steele 1990; Dinning and Schonberg 1991; Netzer and Miller 1991; Mellor-Crummey 1991; Min and Choi 1991; Savage et al. 1997; Cheng et al. 1998]. The most relevant techniques are based, either partially or completely, on static analysis. Some of these techniques concentrate on the analysis of synchronization, and rely on the fact that detecting conflicting accesses is straightforward once the analysis determines which statements may execute concurrently [Taylor 1983; Balasundaram and Kennedy 1989; Duesterwald and Soffa 1991]. Other analyses focus on parallel programs with affine array accesses in loops, and use techniques similar to those from data dependence analysis for sequential programs [Emrath and Padua 1988; Emrath et al. 1989; Callahan et al. 1990]. However, none of these analyses is designed to detect data races in pointer-based multithreaded programs. To the best of our knowledge, the algorithm presented in this paper and its use as the foundation of a symbolic analysis for divide and conquer programs [Rugina and Rinard 2000] is the first attempt to address the static data race detection problem in multithreaded programs that concurrently update shared pointers.

Communication and synchronization optimizations represent another area of related work. The goal in communication optimizations is to develop code transformations that minimize the delays caused by data transfers between threads. These include blocking data into larger pieces before communication operations, caching remote data, or replacing the expensive remote access instructions with efficient local accesses whenever the compiler can detect that the accessed data is thread-local [Krishnamurthy and Yelick 1995, 1996; Zhu and Hendren 1997, 1998]. Compiler optimizations were also developed to reduce the overhead introduced by synchronization operations such as barrier synchronization [Tseng 1995], or mutual exclusion [Diniz and Rinard 1997, 1998; Rinard 1999; Aldrich et al. 1999; Whaley and Rinard 1999: Blanchet 1999; Bogda and Hoelzle 1999; Choi et al. 1999; Ruf 2000]. In pointer-based programs, all of these optimizations require points-to information; they are, at least theoretically, more effective when they use results from precise pointer analyses. However, in the absence of any pointer information, these transformations have to be very conservative about the pointer-based memory accesses. To avoid this situation, some of the above algorithms have used either flow-insensitive pointer analyses or the combined escape and pointer analysis techniques discussed in Section 6.2.

Finally, weak memory consistency models offer better hardware performance, but significantly complicate both developing and optimizing parallel programs. Conceptually, weak consistency models do not guarantee that the writes issued by a thread will be observed in the same order by all the other threads. Shasha and Snir [1998] proposed a compiler analysis and transformation that computes a minimal set of synchronization instructions which prevents the threads from observing reordered writes. Extensions of this algorithm were further developed by other researchers [Krishnamurthy and Yelick 1996; Lee and Padua 2000]. Compilers can therefore apply such transformations to give guarantees about the ordering of write operations both to the programmers and to subsequent compiler optimizations.

7. CONCLUSION

This paper presents a new flow-sensitive, context-sensitive, interprocedural pointer analysis algorithm for multithreaded programs. This algorithm is, to our knowledge, the first flow-sensitive pointer analysis algorithm for multithreaded programs that takes potential interference between threads into account.

We have shown that the algorithm is correct, runs in polynomial time, and, in the absence of interference, produces the same result as the ideal (but intractable) algorithm that analyzes all interleavings of the program statements. We have implemented the algorithm in the SUIF compiler infrastructure and used it to analyze a sizable set of Cilk programs. Our experimental results show that the algorithm has good precision and converges quickly for this set of programs. We believe this algorithm can provide the required accurate information about pointer variables required for further analyses, transformations, optimizations and software engineering tools for multithreaded programs.

APPENDIX

A. FORMAL DEFINITION OF MAPPING AND UNMAPPING

We formalize the mapping and unmapping process as follows. We assume a call site s in a current procedure c that invokes the procedure p. The current procedure c has a set $V_c \subseteq L$ of local variables and a set $F_c \subseteq L$ of formal parameters; the invoked procedure p has a set $V_p \subseteq L$ of local variables and a set $F_p \subseteq L$ of formal parameters. There is also a set $G \subseteq L$ of ghost variables and a set $V \subseteq L$ of global variables. The call site has a set $\{a_1, \ldots, a_n\} = A_s$ of actual parameter location sets; these location sets are generated automatically by the compiler and assigned to the expressions that define the parameter values at the call site. The set $\{f_1, \ldots, f_n\} = F_p$ represents the corresponding formal parameters of the invoked procedure p. Within p, the special location set r_p represents the return value. The location set r_s represents the return value at the call site. Given a points-to graph C, $\text{nodes}(C) = \{l \mid \langle l, l' \rangle \in C \text{ or } \langle l', l \rangle \in C\}$.

The mapping process starts with a points-to information $\langle C, I, E \rangle$ for the program point before the call site. It constructs a one-to-one mapping $m : \text{nodes}(C) \to F_p \cup G \cup V$ that satisfies the following properties:

—$\forall l \in V_c \cup F_c . m(l) \in G - \text{nodes}(C)$.

—$\forall l_1, l_2 \in V_c \cup F_c . l_1 \neq l_2$ implies $m(l_1) \neq m(l_2)$.

—$\forall l \in V \cup (G \cap \text{nodes}(C)) . m(l) = l$.

—$\forall 1 \leq i \leq n . m(a_i) = f_i$.

The following definition extends m to a mapping $\hat{m}$ that operates on points-to graphs.

$$\hat{m}(C) = \{\langle m(l_1), m(l_2) \rangle \mid \langle l_1, l_2 \rangle \in C\}$$

The analysis uses $\hat{m}$ to construct a new analysis context $\langle C_p, I_p \rangle$ for the invoked procedure as defined below. For a set S of nodes and a general points-to graph C, isolated(S, C) denotes all subgraphs of C that are not reachable from S.

$$\begin{aligned} C_p &= (\hat{m}(C) - \text{isolated}(V \cup F_p, \hat{m}(C))) \cup (V_p \cup \{r_p\}) \times \{\texttt{unk}\} \\ I_p &= \hat{m}(I) - \text{isolated}(V \cup F_p, \hat{m}(I)) \end{aligned}$$

The algorithm must now derive an analysis result $\langle C'_p, E'_p, r'_p \rangle$ that reflects the effect of the invoked procedure on the points-to information. It first looks up the new analysis context in the cache of previously computed analysis results for the invoked procedure p, and uses the previously computed result if it finds it. Otherwise, it analyzes p using $\langle C_p, I_p, \emptyset \rangle$ as the starting points-to information. It stores the resulting analysis result $\langle C'_p, E'_p, r'_p \rangle$ in the cache.

The algorithm then unmaps the analysis result back into the naming context at the call site. It constructs an unmapping $u : \text{nodes}(C'_p) \to V_c \cup F_c \cup G \cup V$

that satisfies the following conditions:

—$\forall l \in V_p \cup F_p . u(l) = \mathrm{unk}$.
—$\forall l \in \mathrm{nodes}(C'_p) - ((V_p \cup F_p) \cup \{r_p\}) . u(l) = m^{-1}(l)$.
—$u(r_p) = r_s$.

The following definition extends u to an unmapping $\hat{u}$ that operates on points-to graphs. In the unmapping process below, we use unmappings that map location sets to unk. The definition of $\hat{u}$ removes any resulting edges from unk.

$$\hat{u}(C) = \{\langle u(l_1), u(l_2)\rangle \mid \langle l_1, l_2\rangle \in C\} - \{\mathrm{unk}\} \times L$$

The algorithm uses $\hat{u}$ to create an analysis context $\langle C', I', E'\rangle$ for the program point after the call site:

$$\begin{array}{lcl} C' & = & \hat{u}(C'_p) \cup \hat{u}(\text{ isolated}(G \cup F_p, \hat{m}(C))) \\ I' & = & I \\ E' & = & \hat{u}(E'_p) \cup E \end{array}$$

B. FORMAL TREATMENT AND SOUNDNESS OF THE ANALYSIS

B.1 Parallel Flow Graphs

Definition B.1.1. A *parallel flow graph* is a directed graph $\langle V, E\rangle$ of vertices V and edges $E \subseteq V \times V$. The successor function $\mathrm{succ} : V \to V$ is defined by $\mathrm{suc}(v) = \{u \mid (v, u) \in E\}$ and the predecessor function $\mathrm{pred} : V \to V$ is defined by $\mathrm{pred}(v) = \{u \mid (u, v) \in E\}$. A path p with first vertex v_1 and last vertex v_n is a sequence of vertices $v_1, \ldots, v_n$ such that $\forall_{1 \le i < n}(v_i, v_{i+1}) \in E$. Given two vertices u and v, $[u, v]$ is the set of all paths with first vertex u and last vertex v. A vertex v is *reachable from* a vertex u if $[u, v] \ne \emptyset$.

A parallel flow graph $\langle V, E\rangle$ can have five kinds of vertices: statement vertices, begin vertices, end vertices, parbegin vertices and parend vertices. Statement vertices represent pointer assignments. Begin vertices and end vertices come in corresponding pairs and represent the begin and end of threads. Parbegin and parend vertices come in corresponding pairs and represent the begin and end of a parallel statement block.

Formally, a thread is the set of all vertices between a begin vertex and the corresponding end vertex. We require that each thread be *isolated* by its begin and end vertices, i.e., all edges from outside the thread into the thread point to the thread's begin vertex, and all edges from inside the thread to outside the thread point from the thread's end vertex.

Definition B.1.2. A vertex v is *between* a vertex u and a vertex w if there exists a path from u to w that includes v, i.e. $\exists p \in [u, w] . v \in p$. A set of vertices s is *isolated by* two vertices u and w if $\forall v_1 \in s, v_2 \notin s . (v_1, v_2) \in E$ implies $v_1 = w$ and $(v_2, v_1) \in E$ implies $v_1 = u$.

Definition B.1.3. Given a begin vertex v_b with corresponding end vertex v_e, $\mathrm{thread}(v_b)$ is the set of all vertices between v_b and v_e. We require that $\mathrm{thread}(v_b)$ be isolated by its begin vertex v_b and corresponding end vertex v_e.

Each corresponding pair of parbegin and parend vertices has a set of parallel threads. There is an edge from the parbegin vertex to the begin vertex of each of its threads, and an edge from the corresponding end vertex of the thread to the corresponding parend vertex.

Definition B.1.4. Given a parbegin vertex v_p with corresponding parend vertex v_d, $\forall v_b \in \text{succ}(v_p).v_b$ must be a begin vertex and $\text{pred}(v_b) = \{v_p\}$ and v_b's corresponding end vertex v_e must be an element of $\text{pred}(v_d)$. Also, $\forall v_e \in \text{pred}(v_d).v_e$ must be an end vertex and $\text{succ}(v_e) = \{v_d\}$ and v_e's corresponding begin vertex v_b must be an element of $\text{succ}(v_p)$.

Definition B.1.5. The set of threads of a parbegin vertex v_p is $\{\text{thread}(v_b) \mid v_b \in \text{succ}(v_p)\}$.

The *parallel computation* of a parbegin vertex v_p with corresponding parend vertex v_d is the set of vertices between v_p and v_d. We require that each parallel computation be isolated by its parbegin and parend vertices.

Finally, we require that there be a distinguished begin vertex v_b with corresponding distinguished end vertex v_e such that all vertices $v \in V$ are between v_b and v_e, v_b has no predecessor vertices, and v_e has no successor vertices. All other begin vertices must have a unique parbegin predecessor, and all other end vertices must have a unique parend successor.

Observation B.1.6. V is a thread.

Observation B.1.7. Given two threads t_1 and t_2, either $t_1 \cap t_2 = \emptyset$, $t_1 \subseteq t_2$, or $t_2 \subseteq t_1$. Furthermore, there is a least (under subset inclusion) thread t such that $t_1 \subseteq t$ and $t_2 \subseteq t$.

B.2 Legal Executions

We model legal executions of a parallel flow graph $\langle V, E\rangle$ using sets of sequences of vertices from V. We first define several operators on sequences, then extend these operators to sets of sequences. Given two sequences s_1 and s_2, $s_1;s_2$ is the concatenation of s_1 and s_2, and $s_1||s_2$ is the set of all interleavings of s_1 and s_2.

We extend ; and || to sets of sequences as follows. The vertex v represents the singleton set of sequences with one sequence v, $S_1;S_2 = \cup_{s_1 \in S_1} \cup_{s_2 \in S_2} \{s_1;s_2\}$, and $S_1||S_2 = \cup_{s_1 \in S_1} \cup_{s_2 \in S_2} s_1||s_2$. We define the legal execution sequences of a thread t as follows:

Definition B.2.1. (Legal Executions) Given a thread t with begin vertex v_b and corresponding end vertex v_e, the set of legal executions of t is the smallest set of sequences that satisfies the following recursive definition:

(1) v_b is a legal execution of t.
(2) If $s;v$ is a legal execution of t, $u \in \text{succ}(v)$, and v is not a parbegin vertex, or a thread end vertex, then $s;v;u$ is a legal execution of t.
(3) If $s;v_p$ is a legal execution of t, v_p is a parbegin vertex with corresponding parend vertex v_d, and $s_1, \ldots, s_l$ are legal executions of the l threads of v_p, then all of the sequences in $s;v_p;(s_1|| \cdots ||s_l);v_d$ are legal executions of t.

An *augmentation* of a sequence of vertices with respect to a thread is derived by inserting additional vertices between the vertices of the sequence. The first and last vertices of the augmentation must be the same as in the original sequence. As defined below, none of the additional vertices can be in the thread.

—The only augmentation of v with respect to t is v.

—If s_a is an augmentation of s with respect to t and $v_1; \ldots ; v_n$ is any sequence of vertices not in t (for all $1 \le i \le n$, $v_i \notin t$), then $v; v_1; \ldots ; v_n; s_a$ is an augmentation of $v; s$ with respect to t.

B.3 Representing Analysis Results

Our analysis represents memory locations using a set L of *location sets* in the program. It represents points-to information using points-to graphs $C \subseteq L \times L$.

The analysis generates, for each program point p, a points-to information triple $MTI(p) = \langle C, I, E \rangle$. Here $C \subseteq L \times L$ is the current points-to information, $I \subseteq L \times L$ is the set of interference edges created by parallel threads, and $E \subseteq L \times L$ is the set of edges created by the current thread. We require that $I \subseteq C$. For each vertex v in the flow graph, there are two program points: the point $\bullet v$ before v, and the point $v\bullet$ after v.

We require that the analysis result satisfy the dataflow equations in Figures 3 and 6.

B.4 Soundness

We define soundness by comparing the results of the algorithm presented in the paper with the results obtained by applying the standard pointer analysis algorithm for sequential programs to sequences of vertices from the parallel flow graph. When the functional $[\![\ ]\!]$ (which provides the abstract semantics for statement vertices) is applied to sequences of statement vertices, it generates the same result as the analysis for sequential programs. We therefore use $[\![\ ]\!]$ in the definition of soundness for the analysis for multithreaded programs. To do so, however, we must trivially extend $[\![\ ]\!]$ to non-statement vertices as follows: $[\![v]\!] \langle C, I, E \rangle = \langle C, I, E \rangle$ if v is not a statement vertex.

An analysis result is sound if it is least as conservative as the result obtained by using the standard pointer analysis algorithm for sequential programs on all interleavings of the legal executions.

Definition B.4.1. (Soundness) An analysis result MTI for a flow graph $\langle V, E \rangle$ is *sound* if for all legal executions $v_1; \ldots ; v_n$ of V, it is true that for all $1 \le i \le n$, $C_i^s \subseteq C_i^a$ and $C_{i-1}^s \subseteq C_i^b$, where $C_0^s = L \times \{\texttt{unk}\}$, $\langle C_i^s, I_i^s, E_i^s \rangle = [\![v_1; \ldots ; v_i]\!] \langle \emptyset, \emptyset, \emptyset \rangle$, $\langle C_i^b, I_i^b, E_i^b \rangle = MTI(\bullet v_i)$, and $\langle C_i^a, I_i^a, E_i^b \rangle = MTI(v_i \bullet)$.

Note that in this definition, C_i^s is the result of the sequential analysis after analyzing the first i vertices, C_i^b is the analysis result at the program point before v_i, and C_i^a is the analysis result at the program point after v_i.

B.5 Proof of Soundness

We next prove that any solution to the set of dataflow equations presented in this paper is sound. The proof depends crucially on two key lemmas: the Propagation Lemma, which characterizes the propagation of edges between parallel threads, and the Extension Lemma, which characterizes the propagation of edges into, through, and out of parallel computations.

LEMMA B.5.1. *(Propagation) Let*

—v_p *be a parbegin vertex with threads* $t_1, \ldots, t_n$,
—$v \in t_i$ *be a vertex in one of the threads,*
—$\langle C_v^b, I_v^b, E_v^b\rangle = MTI(\bullet v)$, $\langle C_v^a, I_v^a, E_v^a\rangle = MTI(v\bullet)$,
—$\langle C, I, E\rangle$ *be a points-to information triple, and*
—$e \in L \times L$ *be a points-to edge such that* $c \notin C$ *but* $c \in C'$, *where* $\langle C', I', E'\rangle = [\![\, v \,]\!](C, I, E)$.

Then $C \subseteq C_v^b$ *implies for all* $u \in \cup_{1\leq j\leq n, j\neq i} t_j$, $e \in C_u^b$ *and* $e \in C_u^a$, *where* $\langle C_u^b, I_u^b, E_u^b\rangle = MTI(\bullet u)$ *and* $\langle C_u^a, I_u^a, E_u^a\rangle = MTI(u\bullet)$.

PROOF. If $e \notin C$ but $e \in C'$, then e is in the *gen* set of v for $\langle C, I, E\rangle$. Because the *gen* set of v does not get smaller if the input points-to information gets larger, $C \subseteq C_v^b$ implies e is in the *gen* set of v for $\langle C_v^b, I_v^b, E_v^b\rangle$, which in turn implies that $e \in E_v^a$. The equations will therefore propagate e in the set of edges created by the current thread past sequential merges, through statement vertices, through parallel computations, and past parend vertices to the end vertex of every thread containing v. At each parbegin vertex with a thread t containing v, the equations will therefore propagate e into the incoming set of interference edges for every thread that executes in parallel with t. The equations will therefore propagate e to the current points-to graph for every vertex in every thread that may execute in parallel with a thread containing v. □

Property B.5.1. (Extension) A legal execution s of a thread t has the extension property if for all augmentations $v_1; \ldots; v_n$ of s with respect to t, for all $C_0^s \subseteq C_1^b$ and for all $0 \leq k < j \leq n$ and edges e such that

—$v_j \in t$, and
—$k \neq 0$ implies $v_k \in t$ and $e \notin C_{k-1}^s$, and
—$\forall_{k\leq i<j} . e \in C_i^s$, and
—$\forall_{k\leq i<j} C_i^s \subseteq C_i^a$, and
—$\forall_{k\leq i<j} C_{i-1}^s \subseteq C_i^b$,

it is true that $e \in C_j^b$, where for all $1 \leq i \leq n$,

—$\langle C_i^b, I_i^b, E_i^b\rangle = MTI(\bullet v_i)$,
—$\langle C_i^a, I_i^a, E_i^a\rangle = MTI(v_i\bullet)$, and
—$\langle C_i^s, I_i^s, E_i^s\rangle = [\![\, v_1; \ldots; v_i \,]\!](C_0^s, \emptyset, \emptyset)$.

LEMMA B.5.2. *(Extension) All legal executions s of a thread t have the extension property.*

PROOF. The proof is by structural induction on s; the cases of the proof correspond to the cases in Definition B.2.1.

—**Case 1:** v_b clearly has the extension property.

—**Case 2:** Assume $s;v$ has the extension property, v is not a parbegin or parend vertex, and $u \in \text{succ}(v)$. We must show that $s;v;u$ has the extension property. Let $v_1;\ldots;v_n$ be an augmentation of $s;v;u$ with respect to t such that $v_1;\ldots;v_n$ together with C_0^s, k, j, and e satisfy the antecedent of the extension property. Let i be the index of v in $v_1;\ldots;v_n$ so that $v = v_i$.
If $v_j \neq u$, then we can apply the induction hypothesis to the augmentation $v_1;\ldots;v_i$ of $s;v$ with respect to t to get $e \in C_j^b$. If $v_j = u$, note that $k \leq i < j$, which implies that $e \in C_i^a$. Because $u \in \text{succ}(v)$, $C_i^a \subseteq C_j^b$, and $e \in C_j^b$.

—**Case 3:** Assume $s;v_p$ has the extension property, v_p is a parbegin vertex with corresponding parend vertex v_d and l threads $t_1,\ldots,t_l$. Also assume that $s_1,\ldots,s_l$ are legal executions of the threads $t_1,\ldots,t_l$, and that $s_1,\ldots,s_l$ all have the extension property. We must show that all sequences in $s;v_p;(s_1||\cdots||s_l);v_d$ have the extension property. Let $v_1;\ldots;v_n$ be an augmentation with respect to t of a sequence in $s;v_p;(s_1||\cdots||s_l);v_d$ such that $v_1;\ldots;v_n$ together with C_0^s, k, j, and e satisfy the antecedent of the extension property for s. We must show that $e \in C_j^b$. The proof is a case analysis on the position of v_k and v_j.

  —$k = 0$ or v_k in $s;v_p$ and v_j in $s;v_p$. Then $e \in C_j^b$ by the induction hypothesis.

  —$k = 0$ or v_k in $s;v_p$ and v_j in one of the legal executions of the threads of v_p; assume without loss of generality in s_1. Let v_b be the begin vertex of thread t_1. If $v_j = v_b$, then let i be the index of v_p in $v_1;\ldots;v_n$ so that $v_i = v_p$. Note that $k \leq i < j$ and therefore $e \in C_i^a$. By the dataflow equations for parbegin vertices, $C_i^a \subseteq C_j^b$, which implies $e \in C_j^b$.
  If $v_j \neq v_b$, we will apply the induction hypothesis to s_1. We must therefore find an augmentation, a points-to graph, two integers and an edge that satisfy the antecedent of the extension property.
  Let i be the index in $v_1;\ldots;v_n$ of the begin vertex v_b of t_1 and let h be the index of the corresponding end vertex v_e. Note that $k < i < j \leq h < n$. By the extension property, $C_{i-1}^s \subseteq C_i^b$. The augmentation $v_i;\ldots;v_h$ of s_1 with respect to t_1, the integers 0 and $j - i + 1$ (the index of v_j in $v_i;\ldots;v_h$) and the edge e together satisfy the antecedent of the extension property for s_1. By the induction hypothesis, $e \in C_j^b$.

  —$k = 0$ or v_k in $s;v_p$ and $v_j = v_d$. Then all of the end vertices v_i^e of $s_1,\ldots,s_l$ are between v_k and v_j in $v_1;\ldots;v_n$. By definition of the extension property, for all $1 \leq i \leq l$, $e \in C$, where $\langle C, I, E\rangle = MTI(v_i^e\bullet)$. By definition of the dataflow equations for parend vertices (which intersect the current points-to graphs from the end vertices of the parallel threads), $e \in C_j^b$.

  —v_k and v_j both in the same legal execution of one of the threads of v_p; assume without loss of generality in s_1. We will apply the induction hypothesis to s_1.

Let i be the index in $v_1; \ldots ; v_n$ of the begin vertex v_b of s_1 and let h be the index of the corresponding end vertex v_e. Note that $i \le k < j \le h < n$. We can then apply the induction hypothesis to the augmentation $v_i; \ldots ; v_h$ of s_1 with respect t_1 to obtain $e \in C_j^b$.

—v_k and v_j in different legal executions of the threads of v_p; assume without loss of generality in s_1 and s_2. Then e is in the *gen* set of v_k, and by the Propagation Lemma $e \in C_j^b$.

—v_k in one of the legal executions of the threads of v_p, assume without loss of generality in s_1, and $v_j = v_d$. Then e is in the *gen* set of v_k. For all $1 \le i \le l$, let v_i^e be the end vertices of $s_1, \ldots , s_l$ and let $\langle C_i, I_i, E_i \rangle = MTI(v_i^e \bullet)$. By the Propagation Lemma, for all $2 \le i \le l$, $e \in C_i$. v_1^e is between v_k and v_j in $v_1; \ldots ; v_n$, which implies (by the definition of the extension property) that $e \in C_1$. By the definition of the dataflow equations for parend vertices, $e \in C_j^b$. □

We now prove the soundness theorem.

THEOREM B.5.2. *(Soundness) The dataflow equations presented in this paper are sound.*

PROOF. Given a flow graph $\langle V, E \rangle$, consider any legal execution $v_1; \ldots ; v_n$ of V. Let C_i^s, C_i^b and C_i^a be as in Definition B.4.1. We will show by induction that for all $1 \le j \le n$, it is true that for all $1 \le i \le j$, $C_i^s \subseteq C_i^a$ and $C_{i-1}^s \subseteq C_i^b$.

—**Base Case:** In the initial points-to information all the edges point to unknown, so $C_0^s = C_1^b = L \times \{\mathtt{unk}\}$. Also, since v_1 is a begin vertex, $C_1^a = C_1^b$ and $C_0^s = C_1^s$. Thus, $C_0^s = C_1^s = C_1^a = C_1^b$ and therefore the inclusion relations $C_0^s \subseteq C_0^b$ and $C_1^s \subseteq C_1^a$ are trivially satisfied.

—**Induction Step:** Assume $C_i^s \subseteq C_i^a$ and $C_{i-1}^s \subseteq C_i^b$ for $1 \le i < j$. We must show $C_j^s \subseteq C_j^a$ and $C_{j-1}^s \subseteq C_j^b$. Consider any edge $e \in C_{j-1}^s$. Find the largest $k < j$ such that $e \in C_k^s$ but $e \notin C_{k-1}^s$. If no such k exists, let $k = 0$. By the Extension Lemma applied to $v_1; \ldots ; v_n$, C_0^s, k, j, and e, $e \in C_j^b$, and $C_{j-1}^s \subseteq C_j^b$. By monotonicity of $[\![\;\;]\!]$, $C_j^s \subseteq C_j^a$. □

C. PRECISION OF THE ANALYSIS

We convert a parallel flow graph to a sequential flow graph as follows. Given a par construct with n sequential threads $t_1, \ldots , t_n$, first construct the product graph of the threads. The vertices of this graph are compound vertices of the form $(v_1, \ldots , v_n)$, where v_i is an element of t_i; each edge (v, u) in the original flow graph generates an edge from $(v_1, \ldots , v_{i-1}, v, v_{i+1}, \ldots , v_n)$ to $(v_1, \ldots , v_{i-1}, u, v_{i+1}, \ldots , v_n)$ in the product graph. For each such edge, we construct a new vertex v_u, then replace the edge in the product graph with two edges: one from $(v_1, \ldots , v_{i-1}, v, v_{i+1}, \ldots , v_n)$ to v_u, and another from v_u to $(v_1, \ldots , v_{i-1}, u, v_{i+1}, \ldots , v_n)$. We use the correspondence between v_u and u to relate the program points in the resulting interleaved graph to the program points in the original graph.

The transfer function of the new vertex v_u is the same as the transfer function of u in the original graph; the transfer function of the compound vertices is the identity. The interleaved graph directly represents all possible interleavings of statements from the n parallel threads. We can recursively apply this construction to convert a parallel flow graph to a (potentially much larger) sequential flow graph.

We define interference in the analysis of the interleaved graph as follows. For each vertex u in the original flow graph, find the minimal thread t containing u. There is a set of new vertices v_u in the interleaved graph that correspond to u; during the analysis, label each of the edges in the *gen* set of v_u with the thread t and propagate these edge labels through the analysis. We say that the analysis of a vertex *uses* an edge e in the current points-to graph C if the presence of an edge in the *gen* set as defined in Figure 4 depends on the presence of e in C. We say that there is *interference* if a vertex v_u uses an edge e whose set of labels contains only parallel threads of t. In this case, the only reason e is in the *gen* set of v_u is because some parallel thread created an edge that, in turn, caused v_u to generate e.

If there is no interference in the analysis of the interleaved graph, then we can show that the analysis of each thread in the original graph never uses any of the edges E_j created by other parallel threads. In this case, the direct analysis of par construct terminates in at most two iterations, none of the points-to graphs change during the second iteration, and the analysis result is generated by the interleaving that took place during the first iteration. Since this interleaving is only one of the many interleavings in the interleaved graph, the direct analysis generates a result that is at least as precise as the interleaved analysis.

REFERENCES

ALDRICH, J., CHAMBERS, C., SIRER, E., AND EGGERS, S. 1999. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proceedings of the 6th International Static Analysis Symposium*.

ANDERSEN, L. O. 1994. Program analysis and specialization for the C programming language. Ph.D. thesis, DIKU, University of Copenhagen.

BABB, J., RINARD, M., MORITZ, A., LEE, W., FRANK, M., BARUA, R., AND AMARASINGHE, S. 1999. Parallelizing applications into silicon. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, Napa Valley, CA.

BALASUNDARAM, V. AND KENNEDY, K. 1989. Compile-time detection of race conditions in a parallel program. In *Proceedings of the 1989 ACM International Conference on Supercomputing*, Crete, Greece.

BARUA, R., LEE, W., AMARASINGHE, S., AND AGARWAL, A. 1999. Maps: A compiler-managed memory system for Raw machines. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA.

BLANCHET, B. 1999. Escape analysis for object oriented languages. Application to Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO.

BLIEBERGER, J., BURGSTALLER, B., AND SCHOLZ, B. 2000. Symbolic dataflow analysis for detecting deadlocks in Ada tasking programs. In *Proceedings of the 5th International Conference on Reliable Software Technologies Ada-Europe 2000*.

BOGDA, J. AND HOELZLE, U. 1999. Removing unnecessary synchronization in Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO.

BOGLE, P. AND LISKOV, B. 1994. Reducing cross-domain call overhead using batched futures. In *Proceedings of the 9th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Portland, OR.

Boyapati, C. and Rinard, M. 2001. A parameterized type system for race-free Java programs. In *Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Tampa Bay, FL.

Budiu, M., Goldstein, S., Sakr, M., and Walker, K. 2000. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *Proceedings of the EuroPar 2000 European Conference on Parallel Computing*, Munich, Germany.

Callahan, D., Kennedy, K., and Subhlok, J. 1990. Analysis of event synchronization in a parallel programming tool. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA.

Callahan, D. and Subhlok, J. 1988. Static analysis of low-level synchronization. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, WI.

Carlisle, M. and Rogers, A. 1995. Software caching and computation migration in Olden. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, 29–38.

Chase, D., Wegman, M., and Zadek, F. 1990. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY.

Cheng, G., Feng, M., Leiserson, C., Randall, K., and Stark, A. 1998. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, Puerto Vallarta, Mexico.

Choi, J., Burke, M., and Carini, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual Symposium on Principles of Programming Languages*, Charleston, SC.

Choi, J., Gupta, M., Serrano, M., Sreedhar, V., and Midkiff, S. 1999. Escape analysis for Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO.

Chow, J.-H. and Harrison, W. 1992a. Compile-time analysis of parallel programs that share memory. In *Proceedings of the 19th Annual ACM Symposium on the Principles of Programming Languages*, Albuquerque, NM.

Chow, J.-H. and Harrison, W. 1992b. A general framework for analyzing shared-memory programs. In *Proceedings of the 1992 International Conference on Parallel Processing*, Ann Arbor, MI.

Corbett, J. 1996. Evaluating deadlock detection methods for concurrent software. *IEEE Trans. Softw. Eng. 22*, 3 (Mar.), 161–180.

Cousot, P. and Cousot, R. 1984. *Automatic Program Construction Techniques*. Macmillan Publishing Company, New York, NY.

Das, M. 2000. Unification-based pointer analysis with bidirectional assignments. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada.

Detlefs, D., Leino, K. R., Nelson, G., and Saxe, J. 1998. Extended static checking. Tech. Rep. 159, Compaq Systems Research Center.

Dillon, L. 1990. Using symbolic execution for verification of Ada tasking programs. *ACM Trans. Prog. Lang. Syst. 12*, 4, 643–669.

Diniz, P. and Rinard, M. 1997. Synchronization transformations for parallel computing. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, Paris, France, 187–200.

Diniz, P. and Rinard, M. 1998. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *J. Parallel Distrib. Comput. 49*, 2 (Mar.), 2218–244.

Dinning, A. and Schonberg, E. 1991. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA.

Diwan, A., McKinley, K., and Moss, E. 1998. Tye-based alias analysis. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, Montreal, Canada.

Duesterwald, E. and Soffa, M. 1991. Concurrency analysis in the presence of procedures using a data-flow framework. In *1991 International Symposium on Software Testing and Analysis*, Victoria, Canada.

DWYER, M. AND CLARKE, L. 1994. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, New Orleans, LA.

EMAMI, M., GHIYA, R., AND HENDREN, L. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*, Orlando, FL.

EMRATH, P., GHOSH, S., AND PADUA, D. 1989. Event synchronization analysis for debugging parallel programs. In *Proceedings of Supercomputing '89*, Reno, NV.

EMRATH, P. AND PADUA, D. 1988. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and distributed debugging*, Madison, WI.

FALSAFI, B., LEBECK, A., REINHARDT, S., SCHOINAS, I., HILL, M., LARUS, J., ROGERS, A., AND WOOD, D. 1994. Application-specific protocols for user-level shared memory. In *Proceedings of Supercomputing '94*, Washington, DC, IEEE Computer Society Press, Los Alamitos, Calif., 380–389.

FLANAGAN, C. AND ABADI, M. 1999. Types for safe locking. In *Proceedings of the 1999 European Symposium on Programming*, Amsterdam, The Netherlands.

FLANAGAN, C. AND FREUND, S. 2000. Type-based race detection for Java. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada.

FRIGO, M., LEISERSON, C., AND RANDALL, K. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, Montreal, Canada.

GODEFROID, A. AND WOLPER, P. 1994. A partial approach to model checking. *Information and Computation 110*, 2 (May), 305–326.

GRUNWALD, D. AND SRINIVASAN, H. 1993. Data flow equations for explicitly parallel programs. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA.

HAUSER, C., JACOBI, C., THEIMER, M., WELCH, B., AND WEISER, M. 1993. Using threads in interactive systems: A case study. In *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, Asheville, NC.

HEINTZE, N. AND TARDIEU, O. 2001. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation*, Snowbird, UT.

HICKS, J. 1993. Experiences with compiler-directed storage reclamation. In *Proceedings of the FPCA '93 Conference on Functional Programming Languages and Computer Architecture*, 95–105.

HIND, M. 2001. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the SIGPLAN-SIGSOFT '01 Workshop on Program Analysis for Software Tools and Engineering*, Snowbird, UT.

KNOOP, J. 1998. Code motion for explicitly parallel programs. In *Proceedings of the 4th European Conference on Parallel Processing (Euro-Par'98)*, Southampton, UK.

KNOOP, J. AND STEFFEN, B. 1999. Code motion for explicitly parallel programs. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA.

KNOOP, J., STEFFEN, B., AND VOLLMER, J. 1996. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Program. Lang. Syst. 18*, 3 (May), 268–299.

KRISHNAMURTHY, A. AND YELICK, K. 1995. Optimizing parallel programs with explicit synchronization. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA.

KRISHNAMURTHY, A. AND YELICK, K. 1996. Analyses and optimizations for shared address space programs. *J. Parallel Distrib. Comput. 38*, 2, 130–144.

LANDI, W. AND RYDER, B. 1992. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA.

LEE, J., MIDKIFF, S. P., AND PADUA, D. A. 1998. A constant propagation algorithm for explicitly parallel programs. *Intern. J. Parallel Program. 26*, 5, 563–589.

LEE, J. AND PADUA, D. 2000. Hiding relaxed memory consistency with compilers. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '00)*, Philadelphia, PA.

LEE, J., PADUA, D. A., AND MIDKIFF, S. P. 1999. Basic compiler algorithms for parallel programs. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA.

LONG, D. AND CLARKE, L. 1991. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *1991 International Symposium on Software Testing and Analysis*, Victoria, Canada.

MASTICOLA, S. AND RYDER, B. 1990. Static infinite wait anomaly detection in polynomial time. In *Proceedings of the 1990 International Conference on Parallel Processing*, St. Charles, IL.

MASTICOLA, S. AND RYDER, B. 1993. Non-concurrency analysis. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA.

MELLOR-CRUMMEY, J. 1991. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing '91*. Albuquerque, NM.

MIDKIFF, S. AND PADUA, D. 1990. Issues in the optimization of parallel programs. In *Proceedings of the 1990 International Conference on Parallel Processing*, II–105–113.

MIN, S. AND CHOI, J. 1991. Race Frontier: Reproducing data races in parallel-program debugging. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA.

MORITZ, A., FRANK, M., AND AMARASINGHE, S. 2000. FlexCache: A framework for flexible compiler generated data caching. In *The 2nd Workshop on Intelligent Memory Systems*, Boston, MA.

NAUMOVICH, G. AND AVRUNIN, G. 1998. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of the ACM SIGSOFT '98 Symposium on the Foundations of Software Engineering*, Lake Buena Vista, FL.

NAUMOVICH, G., AVRUNIN, G., AND CLARKE, L. 1999. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proceedings of the ACM SIGSOFT '99 Symposium on the Foundations of Software Engineering*, Toulouse, France.

NETZER, R. AND MILLER, B. 1991. Improving the accuracy of data race detection. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA.

O'CALLAHAN, R. AND JACKSON, D. 1997. Lackwit: A program understanding tool based on type inference. In *1997 International Conference on Software Engineering*, Boston, MA.

REPPY, J. 1992. Higher–order concurrency. Ph.D. thesis, Dept. of Computer Science, Cornell Univ., Ithaca, NY.

RINARD, M. 1999. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Trans. Comput. Syst. 17*, 4 (Nov.), 337–371.

RINARD, M. 2001. Analysis of multithreaded programs. In *Proceedings of the 8th Static Analysis Symposium*, Paris, France.

ROUNTEV, A. AND CHANDRA, S. 2000. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada.

RUF, E. 1995. Context-insensitive alias analysis reconsidered. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA.

RUF, E. 2000. Effective synchronization removal for Java. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada.

RUGINA, R. AND RINARD, M. 1999. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA.

RUGINA, R. AND RINARD, M. 2000. Symbolic bounds analysis of pointers, array indexes, and accessed memory regions. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada.

RYDER, B., LANDI, W., STOCKS, P., ZHANG, S., AND ALTUCHER, R. 2001. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Program. Lang. Syst. 23*, 1 (Mar.), 105–186.

SAGIV, M., REPS, T., AND WILHELM, R. 1998. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst 20*, 1 (Jan.), 1–50.

SALCIANU, A. AND RINARD, M. 2001. Pointer and escape analysis for multithreaded programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Snowbird, Utah.

SARKAR, V. 1997. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In *Proceedings of the Tenth Workshop on Languages and Compilers for Parallel Computing*, Minneapolis, MN.

SAVAGE, S., BURROWS, M., NELSON, G., SOLBOVARRO, P., AND ANDERSON, T. 1997. Eraser: A dynamic race detector for multi-threaded programs. *ACM Trans. Comput. Syst. 15*, 4, 391–411.

SHAPIRO, M. AND HORWITZ, S. 1997. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, Paris, France.

SHASHA, D. AND SNIR, M. 1988. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst. 10*, 2 (Apr.).

SRINIVASAN, H., HOOK, J., AND WOLFE, M. 1993. Static single assignment for explicitly parallel programs. In *Proceedings of the 20th Annual ACM Symposium on the Principles of Programming Languages*, Charleston, SC.

STEELE, G. 1990. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th Annual ACM Symposium on the Principles of Programming Languages*, San Francisco, CA.

STEENSGAARD, B. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL.

STEPHENSON, M., BABB, J., AND AMARASINGHE, S. 2000. Bitwidth analysis with application to silicon compilation. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada.

STERLING, N. 1994. Warlock: A static data race analysis tool. In *Proceedings of the 1993 Winter Usenix Conference*, San Diego, CA.

TAYLOR, R. N. 1983. A general purpose algorithm for analyzing concurrent programs. *Commun. ACM 26*, 5 (May), 362–376.

TSENG, C. 1995. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York, Santa Barbara, CA, 144–155.

VALMARI, A. 1990. A stubborn attack on state explosion. In *Proceedings of the 2nd International Workshop on Computer Aided Verification*, New Brunswick, NJ.

WHALEY, J. AND RINARD, M. 1999. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO.

WILSON, R. AND LAM, M. 1995. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA.

ZHU, Y. AND HENDREN, L. 1997. Locality analysis for parallel C programs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '97)*, San Francisco, CA.

ZHU, Y. AND HENDREN, L. 1998. Communication optimizations for parallel C programs. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, Montreal, Canada.

Received December 1999; revised September 2001; July 2002; accepted August 2002