Pointer Analysis for Multithreaded Programs *

Radu Rugina and Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{rugina, rinard}@lcs.mit.edu

Abstract

This paper presents a novel interprocedural, flow-sensitive, and context-sensitive pointer analysis algorithm for multithreaded programs that may concurrently update shared pointers. 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 full range of constructs in multithreaded programs, including recursive functions, function pointers, structures, arrays, nested structures and arrays, pointer arithmetic, casts between pointer variables of different 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 sizable set of multithreaded programs written in the Cilk multithreaded programming language. Our experimental results show that the analysis has good precision and converges quickly for our set of Cilk programs.

1 Introduction

The use of multiple threads of control is quickly becoming a mainstream programming practice. Programmers use multiple threads for many reasons — 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 [18], and as a general structuring mechanism for large software systems [12].

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. A straightforward adaptation of these techniques, which would analyze all possible interleavings of statements from parallel threads, fails because of a combinatorial explosion in the number of potential program paths.

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 traditional optimizations to multithreaded programs is well known [16]; we believe that the algorithm presented in this paper will move the field much closer to being able to successfully optimize such programs.

The key difficulty associated with 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 may access. 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. 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 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

*This research was supported in part by NSF Grant CCR-9702297.

[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, then merging the results.

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 [10]. The implemented algorithm handles a full range of constructs in multithreaded programs, including function pointers, recursive functions, pointers to structures and arrays, pointer arithmetic, casts between pointer variables of different 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. 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, we show that the algorithm computes a conservative approximation to the result obtained by applying the standard flow-sensitive algorithm to all possible interleaved executions, and 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.

2 Example

Consider the simple multithreaded program in Figure 1. This program first initializes the variables, then, at line 4, uses the `par` construct to create two parallel threads, which execute the statements in lines 5 and 6. Any interleaved execution of these statements may occur when the program runs, and they must both complete before execution proceeds beyond the `par` construct.

The first thread (in line 5) writes a 1 into the memory location to which `p` points. At the beginning of the `par` construct, `p` points to `x`, and it may still point to `x` when the statement `*p = 1` from line 5 executes. But because `q` points to `p`, if the statement `*q = &y` from the thread in line 6 executes before the statement `*p = 1` from line 5, `p` may point to `y` when `*p = 1` executes. Any further analysis or optimization of this program must take both possibilities into account.

When both threads complete, execution proceeds beyond the end of the `par` construct at line 7. At this point, the statement `*q = &y` from line 6 has executed, so `p` definitely points to `y`. The statement `*p=2` at line 8 therefore assigns `y` to 2.

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

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

Figure 1: Example Multithreaded 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.

3 The Algorithm

This section presents the 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* [8]. 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* [25], 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 $\langle \mathtt{v}, 0, 0\rangle$; for a field `f` in a structure `s` is $\langle \mathtt{s}, \mathtt{f}, 0\rangle$ (here `f` is the offset of the field within the structure); for a set of array elements `a[i]` is $\langle \mathtt{a}, 0, \mathtt{s}\rangle$ (here `s` is the size of an element of `a`); and for a set of fields `a[i].f` is $\langle \mathtt{a}, \mathtt{f}, \mathtt{s}\rangle$ (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 $\langle \mathtt{s}, \mathtt{f}, 0\rangle$.

The analysis assumes that if two location sets have different memory block names, they represent disjoint sets of memory locations. This assumption is valid if programs do not violate the array bounds. The analysis also assumes that there are no assignments from integers to pointer variables.

There is a special location set `unk`, which represents the unknown memory location. All pointer variables initially point to the unknown memory location. Dereferencing the unknown location set gives the unknown location set itself, and the assignment of pointer values to the unknown location set through store statements is ignored by the analysis after it issues a warning message. The use of the unknown location set allows us to easily differentiate between *definite* pointers and *possible* pointers. If there is only one edge `x` $\rightarrow$ `y` from the location set `x`, `x` definitely points to `y`. If `x` may point to `y` or it may be uninitialized, there are two edges from `x`: `x` $\rightarrow$ `y` and `x` $\rightarrow$ `unk`.

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

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 *basic pointer assignment statements* in Figure 2:

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

Figure 2: Basic Pointer Assignment Statements

More complicated pointer assignments can be reduced to one or more statements of these basic types. In the above basic statements, `x` and `y` denote location sets, hence an 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 copy assignment 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 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 if 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 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 $[\![\ ]\!] : \mathit{Stat} \rightarrow (P^3 \rightarrow P^3)$ that associates a transfer function $f \in P^3 \rightarrow P^3$ 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).

$$\begin{aligned} \text{deref} &: 2^L \times P \rightarrow 2^L \\ \text{deref}(S, C) &= \{\mathtt{y} \in L \mid \exists \mathtt{x} \in S \,.\, (x, y) \in C\} \end{aligned}$$

The partial order relation $\subseteq$ of points-to graphs can be easily extended to a partial order relation $\sqsubseteq$ over the multithreaded points-to information P^3 as follows:

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

This partial order induces a lattice on P^3; the meet operation in this lattice is the merge operation for the intraprocedural dataflow algorithm:

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

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.

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.

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

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

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

Figure 3: Dataflow Equations for Basic Statements

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

Figure 4: Dataflow Information for Basic Statements

Initial Points-to Edges	Basic Statement	New Points-to Edge
x y	`x = &y`	x ⇢ y
x y → z	`x = y`	x ⇢ y → z
x w y → z	`x = *y`	x ⇢ w y → z
x → w y → z	`*x = y`	x → w y → z

Figure 5: New Points-To Edges for Basic Statements

of the construct until all the child threads complete. It can then execute the following 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.

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 edge `p` → `y` is not present in the points-to graph flowing into the `par` construct, it is present in the flow graph at the beginning of the first thread. This reflects the fact that the assignment `*q = &y` from the second thread may execute before any statement from the first thread. Second, even though the edge `p` → `x` is present in the flow graph at the end of the first thread, it is not present in the flow graph after the `par` construct. This reflects the fact that the second thread always makes `p` point to `y` instead of `x`.

$$[\![\texttt{par}\{\{t_1\}\ldots\{t_n\}\}]\!]\langle C, I, E\rangle = \langle C', I, E'\rangle, \text{ where}$$
$$C' = \bigcap_{1 \le i \le n} C'_i$$
$$E' = E \cup \bigcup_{1 \le i \le n} E_i$$
$$C_i = C \cup \bigcup_{1 \le j \le n, j \ne i} E_j$$
$$I_i = I \cup \bigcup_{1 \le j \le n, j \ne i} E_j$$
$$[\![t_i]\!]\langle C_i, I_i, \emptyset\rangle = \langle C'_i, I_i, E_i\rangle$$

Figure 6: Dataflow Equations for `par` Constructs

We next present an intuitive explanation of the dataflow equations in Figure 6.[2]

[2] The full paper, available at www.cag.lcs.mit.edu/~rinard/paper, provides a formal proof of the soundness of these equations.

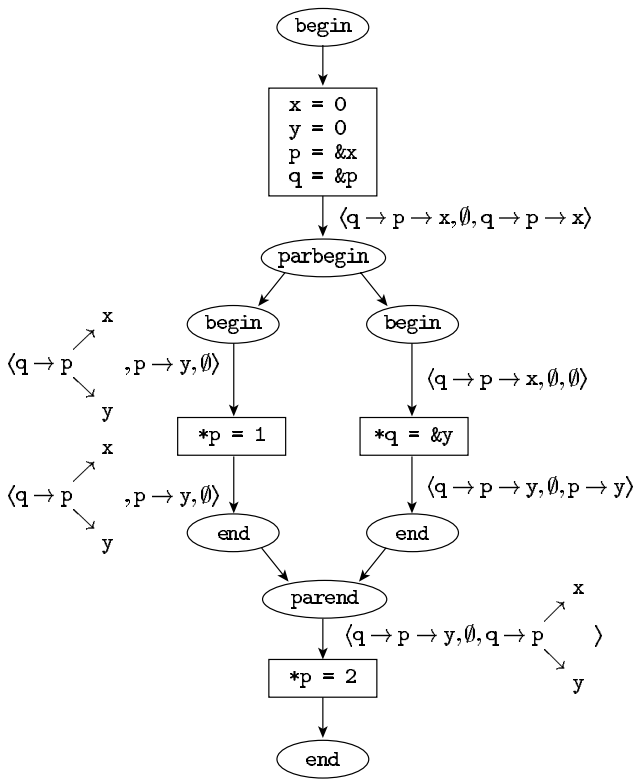

Figure 7: Analysis Result for Example

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.

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 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 parallel flow graph's begin vertex, which 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.

3.6 Complexity of Fixed-Point Algorithm

We derive the following complexity bound. All the location sets that the algorithm manipulates appear in an instruction in the program. The size of the points-to graphs is therefore $O(n^2)$, where n is the number of instructions in the program. Because there is one points-to information for each program point, the maximum size of all of the points-to graphs put together is $O(n^3)$. If the fixed-point algorithm processes n vertices without adding an edge to one of the points-to graphs, it terminates. The algorithm therefore analyzes at most $O(n^4)$ vertices. In practice, we expect that the complexity will be significantly smaller, and our experimental results support this expectation.

3.7 Precision of Analysis

We next discuss the precision of the algorithm relative to an ideal algorithm that analyzes all interleavings of the statements from parallel threads. The basic idea 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 in the interleaved analysis, there is no interference in the direct analysis of the multithreaded program, and the two analyses generate the same result. In this case, our analysis provides a safe, efficient way to obtain the precision of the ideal algorithm.[3]

3.8 Parallel Loops

Parallel loops execute, in parallel, a statically unbounded number of threads that execute the same loop body. 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 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, as described below.

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

The symmetry induced by the identical structure of the concurrent threads produces identical dataflow equations for the threads. The analysis can therefore solve the set of equations for one thread, with the analysis result valid for all threads:

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

The sets C_0' and E_0 defined by the above recursive equation are used to compute the information flowing out of the parallel loop construct:

$$[\![\texttt{parfor(body)}]\!]\langle C, I, E\rangle = \langle C_0', I, E \cup E_0\rangle$$

[3] The full paper, available at www.cag.lcs.mit.edu/~rinard/paper, provides a more detailed discussion.

These equations provide an easy and safe way to model the behavior of the unknown number of concurrent threads spawned by parallel loop constructs.

3.9 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. We handle such variables in the analysis 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.

3.10 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 a technique implemented by Emami, Ghiya and Hendren [8]. 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 [25], 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.

Definition 2 *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* and *invisible variables* used in earlier interprocedural analyses [15, 8]. Ghost location sets serve two purposes: they facilitate the caching of previous analysis results, and they separate the parameters and local variables of the current invocation of a recursive procedure from those of previous invocations, which results in increased precision.

The algorithm sets up an analysis context for the analysis of the invoked procedure as follows.

- 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 maps 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 points-to information. It initializes all of the local variables to point to the unknown location set.

The algorithm analyzes the invoked procedure in this analysis context to get a new context. It unmaps this new context back into the name space of the calling procedure as follows:

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

The analysis continues after the procedure call with this context.

3.10.1 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 $m : \text{nodes}(C) \rightarrow F_p \cup G \cup V$ 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, $\text{isolated}(S, C)$ denotes all subgraphs of C that are not reachable from S.

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

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 $u : \text{nodes}(C'_p) \rightarrow V_c \cup F_c \cup G \cup V$ that satisfies the following conditions:

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

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`.

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

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

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

3.10.2 Recursive Procedures

As described so far, this algorithm does not terminate for recursive procedures. We use a fixed-point approach to eliminate this problem. The algorithm 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.

3.10.3 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.

3.10.4 Function Pointers

The algorithm uses a case analysis to analyze programs with function pointers that may point to more than one function. This algorithm is a straightforward generalization of an earlier algorithm used to analyze serial programs with this property [8]. We also hardwire the behavior of standard library functions into the analysis.

3.11 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 complete.

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.

The primary difficulty that these constructs introduce, however, is the possibility that a child thread may be created on one path to a `sync` statement but not on another. Our analysis handles this situation by adjusting 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 modification ensures that the analysis correctly reflects the possibility that the child thread may not be created and executed.

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 1 presents a list of the programs and several of their characteristics.

[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.

4.1 Benchmark Set

We would like to emphasize the challenging nature of our benchmark set. It includes every benchmark program 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 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. This paper presents, to our knowledge, the first experimental results for the pointer analysis of programs with this amount of recursion, sophisticated pointer-based data structures, and heavy use of pointer arithmetic and casts.

We next discuss the application data in Table 1. 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. 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`.

[5] Available at supertech.lcs.mit.edu/cilk.

[6] Pousse won the program contest associated with the ICFP '98 conference, and was undefeated in this contest.

Program	Lines of Code	Thread Creation Sites	Total(Pointer) Load Instructions	Total(Pointer) Store Instructions	Total(Pointer) Location Sets	Description
barnes	1149	5	352 (318)	161(125)	1289(395)	Barnes-Hut N-body Simulation
block	342	19	140 (140)	9 (9)	546 (64)	Blocked Matrix Multiply
cholesky	932	27	136 (134)	29 (29)	863(100)	Sparse Cholesky Factorization
cilksort	499	11	28 (28)	14 (14)	569 (65)	Parallel Sort
ck	505	2	85 (64)	49 (38)	571 (36)	Checkers Program
fft	3255	48	461 (461)	335(335)	1883(103)	Fast Fourier Transform
fib	53	3	1 (1)	0 (0)	451 (17)	Fibonacci Calculation
game	195	2	9 (8)	9 (8)	508 (38)	Simple Game
heat	360	6	36 (36)	12 (12)	632 (34)	Heat Diffusion on Mesh
knapsack	122	3	14 (14)	6 (6)	444 (21)	Knapsack, Branch and Bound
knary	114	3	4 (4)	0 (0)	473 (20)	Synthetic Benchmark
lu	594	24	16 (16)	13 (13)	688 (87)	LU Decomposition
magic	965	4	83 (83)	74 (74)	739(108)	Magic Squares
mol	4478	33	1880(1448)	595(387)	2324(223)	Viral Protein Simulation
notemp	341	17	136 (136)	6 (6)	546 (64)	Blocked Matrix Multiply
pousse	1379	8	181 (161)	127(118)	905 (88)	Pousse Game Program
queens	106	2	8 (8)	3 (3)	472 (23)	N Queens Program
space	458	23	272 (272)	13 (13)	561 (70)	Blocked Matrix Multiply

Table 1: Cilk Program Characteristics

The precision measurements are complicated by the fact that each procedure may be analyzed in several different points-to contexts. In different contexts, the load or store instructions may require different numbers of location sets to represent the accessed memory location. We therefore count, for each load or store instruction and each context in which the load or store instruction's procedure was analyzed, the number of location sets required to represent the accessed memory location.

We present the precision measures using histograms. 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 8 presents the histogram for all contexts and all load instructions that use pointers to accesss memory in the Cilk programs. Figure 9 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.

Arrays of pointers are one source of potentially uninitialized pointers. 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.

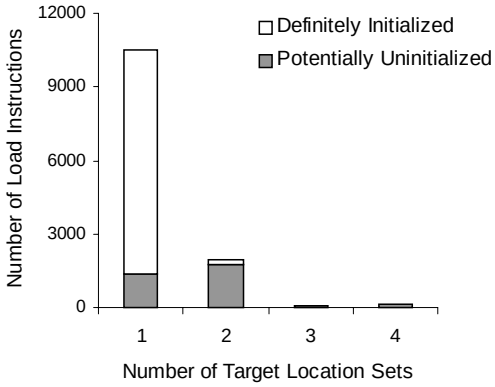

Figure 8: Location Set Histogram for Load Instructions

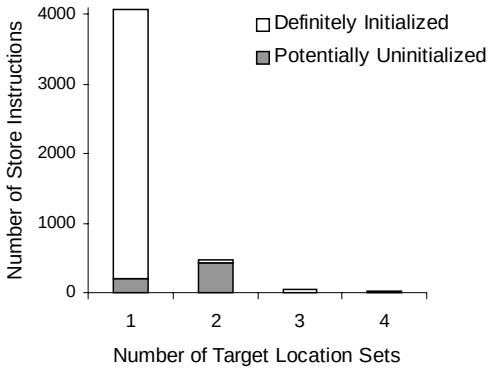

Figure 9: Location Set Histogram for Store Instructions

Program	Load Instructions Number of Read Location Sets 1	2	3	4	Store Instructions Number of Written Location Sets 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 2: Per-Program Counts of the Number of Location Sets Required to Represent an Accessed Location – Separate Contexts, with Ghost Location Sets

NULL pointers are another source of potentially uninitialized pointers. In our analysis, NULL points to 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.

Table 2 breaks the counts down for each application. 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.

4.3 Analysis Measurements

To give some idea of the complexity in practice of the analysis, 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 parallel loops and continuation threads (see Section 3.11) 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 and each parallel loop, as well as the synchronization points that wait for continuation threads. Table 3 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.

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

Table 3: Analysis Measurements

	Load Instructions Number of Read Location Sets					Store Instructions 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)	-	-	-

Table 4: 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

4.4 Comparison with Sequential Analysis

Ideally, we would evaluate the precision in practice 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 interleavings of statements from parallel threads. But the intractability of the Interleaved algorithm makes it impractical to use this algorithm even for comparison purposes.

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

It is important to realize that the appropriate precision metric 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.

But 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 [2]. 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.

We present a precision metric that is appropriate for optimizations, such as the instruction scheduler mentioned above, that use all of the contexts in each procedure. We instrumented the analysis to record, for each ghost location set, the actual location sets in the program that were mapped to the ghost location set. This information allows us to compute, for each load or store instruction that uses a pointer to access memory, the number of actual location sets required to represent the accessed memory location. Table 4 presents this data for the Multithreaded algorithm. Like Table 2, this table provides parenthesized counts of the number of instructions that dereference a potentially uninitialized pointer. There are two differences between the data in these two tables:

- Table 2 counts the location sets for each instruction multiple times: once for each analysis context. Table 4 merges the analysis contexts, then counts the location sets once for each instruction.
- If a ghost location set represents the accessed memory location, Table 2 counts the ghost location set. Table 4 counts the actual location sets that were mapped to that ghost location set during the analysis.

Even though the Sequential and Multithreaded analysis algorithms generate different analysis contexts, they produce virtually identical location set counts after the contexts are merged and the ghost location sets are replaced by the corresponding actual location sets. In fact, the counts are identical for all applications except pousse. 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 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.

Program	Sequential Analysis Times	Multithreaded Analysis Times
barnes	496.73	509.91
blockedmul	11.21	13.89
cholesky	86.40	229.65
cilksort	3.56	3.99
ck	2.7	2.90
fib	0.08	0.11
fft	50.52	142.48
game	1.59	1.74
heat	4.40	8.97
knapsack	0.32	0.34
knary	0.17	0.24
lu	6.36	7.63
magic	26.29	29.35
mol	1540.70	1823.37
notempmul	6.36	8.96
pousse	45.52	67.34
queens	1.17	2.47
space	22.63	27.70

Figure 10: Analysis Times for Sequential and Multithreaded Analysis Algorithms

4.5 Analysis Times

Table 10 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 and Immediately Envisioned Uses

To date, our pointer analysis algorithm has been used in two projects: an instruction scheduling project and an automatic parallelization project. The MIT RAW compiler uses our analysis to disambiguate the targets of load and store instructions [2]. The goal is to exploit instruction-level parallelism and to determine statically which memory modules specific instructions may access.

We have used the pointer analysis results as a foundation for the symbolic analysis and parallelization of divide and conquer algorithms [20]. For efficiency reasons, these 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.

Both of these projects use the pointer analysis algorithm only on sequential programs. In the near future, we plan to build a static race detector and symbolic array bounds checker for multithreaded implementations of divide and conquer algorithms. This project will use the algorithm on multithreaded programs.

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 [21, 5]. 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. The tool could use the pointer analysis information to generate, for each read or write, the set of potentially accessed memory locations. Correlating loads and stores from parallel threads would quickly identify statements that could interfere. Programmers could use the 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 a tool would 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 [7, 3, 26]. 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, we intend to apply pointer analysis to 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 [4]. 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 [9].

6 Related Work

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

6.1 Pointer Analysis for Sequential Programs

Pointer analysis for sequential programs is a relatively mature field [22, 23, 19, 25, 1, 8, 6, 15]. We classify analyses with respect to two properties: flow sensitivity and context sensitivity. Flow-sensitive analyses take the statement ordering into account, and typically use dataflow analysis to produce a points-to graph or set of alias pairs for each program point [19, 25, 8, 6, 15]. 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 [1, 23, 22, 17]. 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 [24], 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 of 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 [19, 25].

A specific kind of imprecision in the analysis of recursive procedures makes many pointer analysis algorithms unsuitable for our purposes. We intend to use 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. To our knowledge, the only published analyses that satisfy this requirement even for sequential programs are both flow sensitive and use some variant of the concept of invisible variables [15, 8, 25]. Other context-sensitive analyses merge information from recursive call sites in a way that destroys the distinction between multiple instantiations of the same variable in a recursive procedure [17], although a flow-insensitive, constraint-based analysis with polymorphic recursion may be able to generate sufficiently precise results.

6.2 Analysis of Multithreaded Programs

Unlike pointer analysis of sequential programs, the analysis of multithreaded programs is a relatively unexplored field. There is an awareness that multithreading significantly complicates program analysis [16], but a full range of standard techniques have yet to emerge. Grunwald and Srinivasan present a dataflow analysis framework for reaching definitions for explicitly parallel programs [11], and Knoop, Steffen and Vollmer present an efficient dataflow analysis framework for bit-vector problems such as liveness, reachability and available expressions, but neither framework applies to pointer analysis [14]. In fact, the application of these frameworks for programs with pointers would require pointer analysis information. Zhu and Hendren present a set of communication optimizations for parallel programs that use information from their pointer analysis; this analysis uses a flow-insensitive analysis to detect pointer variable interference between parallel threads [26]. Hicks also has developed a flow-insensitive analysis specifically for a multithreaded language [13].

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.

References

[1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[2] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: A compiler-managed memory system for Raw machines. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA, May 1999.

[3] P. Bogle and B. Liskov. 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, October 1994.

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

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

[6] J. Choi, M. Burke, and P. Carini. 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, January 1993. ACM.

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

[8] M. Emami, R. Ghiya, and L. Hendren. 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*, pages 242–256, Orlando, FL, June 1994. ACM, New York.

[9] B. Falsafi, A. Lebeck, S. Reinhardt, I. Schoinas, M. Hill, J. Larus, A. Rogers, and D. Wood. Application-specific protocols for user-level shared memory. In *Proceedings of Supercomputing '94*, pages 380–389, Washington, DC, November 1994. IEEE Computer Society Press, Los Alamitos, Calif.

[10] M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, Montreal, Canada, June 1998.

[11] D. Grunwald and H. Srinivasan. 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, May 1993.

[12] C. Hauser, C. Jacobi, M. Theimer, B. Welch, and M. Weiser. Using threads in interactive systems: A case study. In *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, Asheville, NC, December 1993.

[13] J. Hicks. Experiences with compiler-directed storage reclamation. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 95–105, June 1993.

[14] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, May 1996.

[15] W. Landi and B. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[16] S. Midkiff and D. Padua. Issues in the optimization of parallel programs. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II–105–113, 1990.

[17] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *1997 International Conference on Software Engineering*, Boston, MA, May 1997.

[18] J. Reppy. *Higher–order Concurrency.* PhD thesis, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., June 1992.

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

[20] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.

[21] S. Savage, M. Burrows, G. Nelson, P. Solbovarro, and T. Anderson. Eraser: A dynamic race detector for multi-threaded programs. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[22] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, Paris, France, January 1997.

[23] Bjarne Steensgaard. 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, January 1996.

[24] P. Stocks, B. Ryder, W. Landi, and S. Zhang. Comparing flow and context sensitivity on the modification side-effects problem. In *1998 International Symposium on Software Testing and Analysis*, Clearwater, FL, March 1998.

[25] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995. ACM, New York.

[26] H. Zhu and L. Hendren. Communication optimizations for parallel C programs. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, Montreal, Canada, June 1998.