Parallel Replication-based Points-to Analysis

Sandeep Putta, Rupesh Nasre

Indian Institute of Technology, Bombay, India,
Indian Institute of Science, Bangalore, India.
sandeep.p@iitb.ac.in,nasre@csa.iisc.ernet.in

Abstract. Pointer analysis is one of the most important static analyses during compilation. While several enhancements have been made to scale pointer analysis, the work on parallelizing the analysis itself is still in infancy. In this article, we propose a parallel version of context-sensitive inclusion-based points-to analysis for C programs. Our analysis makes use of *replication* of points-to sets to improve parallelism. In comparison to the former work on parallel points-to analysis, we extract more parallelism by exploiting a key insight based on monotonicity and unordered nature of flow-insensitive points-to analysis. By taking advantage of the nature of points-to analysis and the structure of constraint graph, we devise several novel optimizations to further improve the overall speed-up. We show the effectiveness of our approach using 16 SPEC 2000 benchmarks and five large open source programs that range from 1.2 KLOC to 0.5 MLOC. Specifically, our context-sensitive analysis achieves an average speed-up of 3.4× on an 8-core machine.

1 Introduction

Points-to analysis [1, 28, 6, 4, 17] is a method of statically determining whether two pointers may point to the same location at runtime. The two pointers are then said to be aliases of each other. While pointer analysis is immensely helpful for compiler optimizations and extracting parallelism, the analysis itself can be run in parallel to take advantage of the multiple resources available.

There is very little literature on parallelizing pointer analysis. Kahlon [17] proposed *bootstrapping* that identifies alias sets using Steensgaard's analysis [28] and then Andersen's analysis [1] is *simulated* to run in parallel on each alias set. Lojo et al. [20] proposed speculative parallelization of inclusion-based pointer analysis to expose amorphous data-parallelism in C programs. Recently, Edvinsson et al. [7] proposed parallel points-to analysis for multi-core machines by exploiting the independence of polymorphic calls and control-flow branches of Java programs. Our method finds more fine-grained parallelism compared to the above methods.

A typical method of parallelizing points-to analysis involves two tasks: (i) identifying non-conflicting constraints and (ii) analyzing the non-conflicting constraints in parallel. The parallelism extracted by the existing techniques is limited due to the inherent irregular nature of the applications (e.g., several SPEC 2000 benchmarks). For instance, speedup in parallel online analysis time in Lojo et al.'s work [20] is maximum 2x using 8-cores over a set of open source C programs analyzed, while in Edvinsson et al.'s work [7] it is maximum 1.76 using 8-cores over a set of Java benchmarks. Thus, we see that irregularity of applications (as defined in [20]) has been a stumbling block while extracting parallelism for performing points-to analysis.

We observe that the monotonicity of a flow-insensitive points-to analysis can help us achieve more parallelism. A key insight is that by allowing the analysis to keep multiple copies of points-to sets, one can reduce dependence across constraints. In fact, with sufficient number of copies, task (i) above gets trivialized since no two constraints conflict! This allows us complete flexibility while scheduling constraints on multiple cores. On the downside, this kind of data replication can affect analysis soundness, as certain data flow facts may not get computed. However, by exploiting monotonicity property of a flow-insensitive analysis, we show that the analysis soundness can be preserved by carefully merging the multiple copies in each iteration of the analysis. This helps us develop a replication-based, yet sound, parallel analysis.

Major contributions of this paper are as below.

- A replication-based data flow analysis to improve parallelism and a novel method based on monotonicity and unordered nature of flow-insensitive algorithm to achieve a sound analysis in the presence of replicated data.
- Instantiating the data-flow analysis to develop a replication-based parallel points-to analysis.
- Several engineering optimizations specific to pointer analysis, like parallel online cycle elimination and limited cycle detection for extracting more parallelism and for a scalable implementation.
- Detailed evaluation of our method using SPEC 2000 benchmarks and five large open source programs (*httpd*, *sendmail*, *ghostscript*, *gdb* and *wine-server*). Our context-sensitive (insensitive) inclusion-based parallel version achieves an average speed-up of 3.4x (3.0x) on an 8-core machine.

2 Motivation and Background

While our approach applies to several data-flow analyses, we illustrate it by parallelizing Andersen's inclusion-based analysis [1]. Thus, we deal with flow-insensitive points-to analysis. For analyzing a general purpose C program, it is sufficient to consider all pointer statements of the following forms: address-of assignment ($\mathtt{p} = \&\mathtt{q}$), copy assignment ($\mathtt{p} = \mathtt{q}$), load assignment ($\mathtt{p} = *\mathtt{q}$) and store assignment ($*\mathtt{p} = \mathtt{q}$) [24]. Address-of constraints can be evaluated only once. Thus, the analysis iterates over the other three kinds of constraints until a fixed-point.

Consider the following example program.

$\mathtt{p} = \&\mathtt{a}, \mathtt{a} = \&\mathtt{x}, \mathtt{b} = \&\mathtt{y}, \mathtt{c} = \&\mathtt{z}, \mathtt{d} = \&\mathtt{w}, \mathtt{q} = \mathtt{p}, \mathtt{a} = \mathtt{b},$
$\mathtt{e} = \mathtt{a}, \mathtt{r} = \mathtt{q}, \mathtt{a} = \mathtt{c}, \mathtt{s} = \mathtt{r}, \mathtt{e} = *\mathtt{a}, \mathtt{t} = \mathtt{s}, \mathtt{a} = \mathtt{d}, *\mathtt{e} = \mathtt{a}$

Statement	Iteration 0	Iteration 1	Iteration 2	Iteration 3
p = &a	p→{a}			
a = &x	a→{x}			
b = &y	b→{y}			
c = &z	c→{z}			
d = &w	d→{w}			
q = p		q→{a}		
a = b		a→{y}		
e = a		e→{x,y}	e→{z,w}	
r = q		r→{a}		
a = c		a→{z}		
s = r		s→{a}		
e = *a				
t = s		t→{a}		
a = d		a→{w}		
*e = a		x,y→{x,y,z,w}	z,w→{x,y,z,w}	

Table 1: Running example (13 steps: 10 for Iteration 1 and 3 for Iteration 2)

A sequential analysis of the above program is given in Table 1. Iteration 0 shows the initial points-to information after processing address-of constraints. We denote points-to information as a set of variables (e.g., {x,y}) and the points-to relation using an arrow (→). The analysis requires three iterations to reach a fixed-point which contains the following points-to facts.

$\mathtt{p}, \mathtt{q}, \mathtt{r}, \mathtt{s}, \mathtt{t} \rightarrow \{\mathtt{a}\}, \mathtt{a}, \mathtt{e}, \mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{w} \rightarrow \{\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{w}\},$
$\mathtt{b} \rightarrow \{\mathtt{y}\}, \mathtt{c} \rightarrow \{\mathtt{z}\}, \mathtt{d} \rightarrow \{\mathtt{w}\}$

For exposition purpose, we define a *step* as a computation of points-to information of a variable. Intuitively, it is proportional to the amount of time required to analyze a constraint. Thus, each copy and load constraint requires a single step, while each store constraint requires zero or more steps depending upon the points-to information of the dereferenced variable. Thus, the above example requires 13 steps (of non-address-of constraints) to reach a fixed-point for a sequential analysis. It is easy to see that the non-address-of constraints in the example require at least 12 steps to compute the fixed-point.

Analyzing a constraint `DST = SRC` involves (i) reading points-to information of SRC and (ii) updating points-to information of DST. Depending upon the type of the constraint, we have different read and write sets as shown in Table 2. The read-set for the load constraint (p = *q) contains not only the pointees of q but also q itself. Similarly, the read-set of the store constraint (*p = q) contains both p and q. Due to weak-typing of C language, read and write sets for a constraint need not be mutually exclusive. For instance, following statement is valid (with type-casting): p = *p, and its read-set as well as write-set contain p.

Two constraints C1 and C2 *conflict* with each other if at least one of the following three conditions holds: (i) ReadSet(C1) $\cap$ WriteSet(C2) $\neq \phi$ (ii) WriteSet(C1) $\cap$ ReadSet(C2) $\neq \phi$ (iii) WriteSet(C1) $\cap$ WriteSet(C2) $\neq \phi$.

Constraint	Read Set	Write Set
p = q	{q}	{p}
p = *q	$\{q\} \cup \{x : q \rightarrow \{x\}\}$	{p}
*p = q	{q, p}	$\{x : p \rightarrow \{x\}\}$

Table 2: Read-Write sets for points-to constraints

The *conflict* relation is reflexive, symmetric and non-transitive. The above conditions can be easily generalized to multiple constraints. Non-conflicting constraints can be analyzed in parallel. Therefore, identifying the constraints that can be evaluated in parallel involves computing read and write (RW) sets of various constraints in some form. As more points-to information gets computed, the read sets of load constraints and the write sets of store constraints also go on increasing in size, potentially reducing opportunities for parallelism as the analysis progresses. Due to monotonic nature of the flow-insensitive analysis, the RW sets never shrink. Note that since the points-to sets are computed dynamically, the RW sets need to be updated dynamically. Using the RW sets, one can parallelize analyzing the example as shown in the parallel schedule below.

Thread T1: q = p, r = q, s = r, t = s
Thread T2: a = b, a = c, a = d, e = *a, e = a, *e = a

The parallel schedule involves analyzing the two sets of constraints using two threads T1 and T2. Compared to 12 (minimum number of) steps of the sequential analysis, the parallel analysis requires at most 9 steps to reach the fixed-point: one step each for the first five copy and load constraints of T2 and four steps for the last store constraint of T2 as pointer e points to four variables {x, y, z, w}. However, due to conflicting constraints, it is not possible to take advantage of more than two cores for this example using a naive parallel points-to analysis. Thus, even if a points-to analysis is provided with four or eight cores, the parallel analysis would still require at least 9 steps. Further, computing and maintaining RW sets, identifying conflicts across constraints and generating a parallel schedule is a time-consuming process. As the RW sets of a constraint change, it needs to be checked for conflict against all other non-conflicting constraints. This checking requires O(n^2) operations *per change in the read-write set* where n is the number of constraints. Conflicting constraints are commonplace [13] and therefore, a naive parallel analysis is going to be prohibitive in terms of not only the amount of parallelism but also the parallel execution time. Our technique illustrates how to extract more parallelism efficiently even when several conflicting constraints exist.

3 Replication-based Analysis

In this section we first introduce our replication-based approach at a higher level by giving an outline of the algorithm. We prove that a replication-based approach is sound for a monotonic, unordered data-flow analysis (e.g., a flow-insensitive points-to analysis). We then explain in detail how replication works for our parallel points-to analysis algorithm.

A flow-insensitive analysis ignores the control-flow information and assumes that program statements may be executed in any order. We state a well-known property of flow-insensitive data-flow analysis to prove soundness of our method.

Theorem 1. *A flow-insensitive data-flow analysis computes the same fixed-point irrespective of the order in which the program statements are analyzed.*

3.1 Algorithm Outline

A data-flow analysis is monotonically increasing if it never *kills* a computed data-flow fact. A flow-insensitive points-to analysis is an example of a monotonically increasing (or simply, monotonic) analysis whereas a flow-sensitive points-to analysis is an example of a non-monotonic analysis. Since flow-insensitive points-to analysis is monotonic and its solution does not depend upon the order of evaluation, our method partitions the set of constraints arbitrarily, analyzes them in parallel, merges the individual solution sets and repeats this process until a fixed-point, as shown in Algorithm 1. The merge operation performs a union of points-to information for each pointer. Thus, if thread T1 computes the following points-to information: p $\rightarrow$ {a}, q $\rightarrow$ {b}, and thread T2 computes the following points-to information: p $\rightarrow$ {c}, q $\rightarrow$ {a,b}, r $\rightarrow$ {c}, then the merge operation on Line 5 of Algorithm 1 computes the following points-to information: p $\rightarrow$ {a,c}, q $\rightarrow$ {a,b}, r $\rightarrow$ {c}.

We first prove that the parallel algorithm computes a safe solution. It is sufficient to prove that the algorithm computes the same solution as that computed by a sequential analysis.

Theorem 2. *Algorithm 1 is sound.*

Proof. Let SEQ be the least fixed-point points-to set computed by a flow-insensitive sequential analysis. We prove that Algorithm 1 computes a solution PAR which equals SEQ, i.e., SEQ = PAR. In this proof, we make a simplistic assumption that in every iteration of the parallel analysis, the constraints are executed in the same order. A multithreaded schedule of the input points-to constraints C can then be represented as an interleaving of the constraints, with constraints analyzed in parallel placed in an arbitrary (but fixed) order. This interleaving forms a sequential ordering S over all the constraints and since it is derived from the parallel schedule, its least fixed-point solution SEQ′ must equal PAR, i.e., SEQ′ = PAR. Now, assume a sequential flow-insensitive analysis analyzing C in the order denoted by S. By Theorem 1, the least fixed-point solution SEQ′ computed by this sequential analysis must equal SEQ, i.e., SEQ = SEQ′. Therefore, SEQ = PAR, proving that Algorithm 1 computes the same solution as that computed by a sequential analysis.

Algorithm 1 Outline of our parallel points-to analysis

```
Require: set C of points-to constraints, number of cores N
 1: partition C arbitrarily into N threads
 2: repeat
 3:     schedule N threads on N cores
 4:     wait for N threads to complete
 5:     merge points-to sets of N threads
 6: until fixed-point
```

We would like to emphasize that both monotonicity and unorderedness are the required properties of the underlying data-flow analysis for our parallel algorithm to compute a safe result. Our algorithm is general and is applicable to any analysis that is monotonic and unordered. For instance, it can be applied to 0-CFA [25].

3.2 Replication

Our method handles conflicting constraints by keeping multiple copies of the conflicting variables and their associated points-to sets. For instance, the constraints p = q, r = p and p = s conflict. However, if they are analyzed in separate threads T1, T2 and T3 respectively, our method creates a copy of p's points-to set in T1 and T3 since their write sets contain p. Since T2 only reads p, it continues to read the master (original) copy of p. The newly computed points-to information of T1 and T3 is merged with that of the master copy of p at the end of each iteration. Note that T2 would contain a copy of the points-to set of r. Further note that it is possible to use multiple copies of variables since the analysis is monotonic. If the analysis is non-monotonic (for instance, if it is flow-sensitive) then naively making multiple copies of the points-to sets may not preserve the solution.

The merge operation performs a union of points-to information for each pointer. Merging of the points-to information for multiple pointers is done in parallel. However, merging of local points-to information of a pointer with its master copy is done in a sequential manner.

Table 3 shows the parallel analysis of the example program using our method. Recall from Section 2 that a naive parallel analysis using read-write sets could not take advantage of more than two cores. Therefore, we illustrate our technique using three threads. Column 1 shows the thread number. Column 2 shows the points-to constraints assigned to each thread. Column 3 and 4 show the new points-to information created as a local copy and the merging of the local copies with the master copies for Iteration 1. The local copies of variable v are denoted as v', v", ... Further columns show the analysis for further iterations.

In contrast to three iterations of the sequential analysis, the parallel analysis requires four iterations to reach the fixed-point. This happens because some points-to information computed by the constraints partitioned across different

T	Stmt	Itr 1	Merge 1	Itr 2	Merge 2	Itr 3	Merge 3	Itr 4
1	q = p	q'→{a}						
	a = b	a'→{y}						
	r = q	r'→{a}						
	e = a	e'→{x,y}	a→{y,z,w},	e'→{x,y,z,w}	e→{z,w}		z,w→{x,y,z,w}	
2	a = c	a"→{z}	e→{x,y}		x,y→{x,y,z,w}		t→{a}	
	e = *a		q,r→{a}		s→{a}			
	s = r			s'→{a}				
3	a = d	a"'→{w}						
	t = s					t'→{a}		
	*e = a			x',y'→{x,y,z,w}		z',w'→{x,y,z,w}		

Table 3: Parallel analysis using three threads (12 steps): Iterations 1, 2, 3 require 4, 2, 3 steps respectively and each merge requires 1 step

T	Stmt	Itr 1	Merge 1	Itr 2	Merge 2	Itr 3	Merge 3	Itr 4
1	q = p	q'→{a}						
	e = a	e'→{x}		e'→{y,z,w}				
2	r = q		a→{y,z,w},	r'→{a}	e→{y,z,w}		y,z,w→{x,y,z,w}	
	a = c	a"→{z}	e→{x},		x→{x,y,z,w}		s,t→{a}	
3	e = *a		q→{a}		r→{a}			
	s = r							
	t = s					s',t'→{a}		
	a = d	a"'→{w}						
4	*e = a			x'→{x,y,z,w}		y',z',w'→{x,y,z,w}		
	a = b	a'→{y}						

Table 4: Parallel analysis using four threads (9 steps): Iterations 1, 2, 3 require 2, 1, 3 steps respectively and each merge requires 1 step

threads requires another iteration to propagate. In general, our multi-copy parallel analysis requires upto 30% more number of iterations over its sequential counterpart. However, by utilizing more cores and extracting more parallelism by making copies, the overall parallel analysis time gets much smaller.

Observe from Table 3 that our parallel analysis requires 12 steps (or time-units) to reach the fixed-point (4 for Iteration 1, 2 for Iteration 2 and 3 for Iteration 3). Since merge operation for multiple pointers is done in parallel, we add one step for each merge operation. If we increase the number of threads to four, our analysis computes the same fixed-point in 9 steps (Table 4: 2 for Iteration 1, 1 for Iteration 2, 3 for Iteration 3 and 3 for the merge operations). If we further increase the number of threads to five, our analysis can compute the fixed-point in 8 steps (Table 5: 1 for Iteration 1, 1 for Iteration 2, 3 for Iteration 3 and 3 for the merge operations). Recall from Section 2 that the sequential version required at least 12 steps to reach the fixed-point and the naive parallel version required 9 steps. This illustrates the unique ability of our method to extract more and more fine-grained parallelism from a seemingly sequential component of a program and improving the resource usage of multiple cores.

Note also that replication of points-to information is not transparent to the threads. Each thread simply deals with its own copy whenever it modifies data. A thread does not need to know about other threads or the number of copies of the points-to information it accesses in the system. This helps in keeping the multithreaded code simple and greatly eases the code understanding. We

T	Stmt	Itr 1	Merge 1	Itr 2	Merge 2	Itr 3	Merge 3	Itr 4
1	e = a	e'→{x}		e'→{y,z,w}				
2	q = p	q'→{a}						

3	r = q
	a = c
		a''→{z}	a→{y,z,w},
			e→{x},
			q→{a}
				r'→{a}	e→{y,z,w}
					x→{x,y,z,w}
					r→{a}
							y,z,w→{x,y,z,w}
							s,t→{a}	

4	e = *a s = r t = s a = d	a'''→{w}				s',t'→{a}		
5	*e = a a = b	a'→{y}		x'→{x,y,z,w}		y',z',w'→{x,y,z,w}		

Table 5: Parallel analysis using five threads (8 steps): Iterations 1, 2, 3 require 1, 1, 3 steps respectively and each merge requires 1 step

would like to emphasize that this property is an artifact of the monotonicity and unorderedness of flow-insensitive analysis.

4 Parallel Points-to Analysis Algorithm

In this section we discuss our parallel points-to analysis algorithm in more detail. Next, we discuss key optimizations which improve overall parallelism of the analysis.

Each analyzer thread runs Algorithm 2. The scheduler (parent) thread communicates with the analyzer threads using global variables $mystate_i$. Each thread runs an indefinite loop until fixed-point (Lines 1–33). The fixed-point is determined by the parent thread during merging operations (Line 5). When a thread is scheduled with an input set of constraints, it analyzes each constraint and depending upon the type of the constraint, it creates local copies of the write-sets and updates the points-to information locally. Note that the local copies are not erased at the end of an iteration and therefore get automatically cached for the further iterations by the thread. As more points-to information is computed, additional local copies of variables are created by the thread. Lines 13–15 process a load constraint (p = *q), The for-loop at Lines 22–24 process a store constraint (*p = q) and Line 29 process a copy constraint (p = q). After processing all the input constraint, Thread i updates its state in global variable $mystate_i$.

The parent thread spawns analyzer threads, waits for them to complete an iteration each, merges their local copies of points-to sets with the master copy in parallel and checks if the fixed-point is reached. Although not shown in the Algorithm, accesses to the global variables (*fixed-point* and $mystate_i$) are protected using locks. Since these accesses occur infrequently (once per thread per iteration), these do not affect the analysis performance in any significant manner.

4.1 Load Balancing

Replication allows an arbitrary distribution of constraints to threads. However, to achieve good performance, proper load-balancing of work is necessary. Unfortunately, the amount of points-to information propagated from one pointer to another differs significantly across pointers and across iterations. Therefore,

Algorithm 2 Points-to analysis by thread i.

```
Require: thread id $i$, set $C_i$ of points-to constraints
 1: while true do
 2:    while $mystate_i$ = I do not have work do
 3:       ;
 4:    end while
 5:    if fixed-point reached then
 6:       break;
 7:    end if
 8:    for each constraint c $\in C_i$ do
 9:       if c is a load constraint p = *q then
10:          if p is not copied locally then
11:             make a local copy p' of p
12:          end if
13:          for each v $\in$ points-to set of q do
14:             points-to set(p') $\cup=$ points-to set(v)
15:          end for
16:       else if c is a store constraint *p = q then
17:          for each v $\in$ points-to set(p) do
18:             if v is not copied locally then
19:                make a local copy v' of v
20:             end if
21:          end for
22:          for each v $\in$ points-to set(p) do
23:             points-to set(v') $\cup=$ points-to set(q)
24:          end for
25:       else if c is a copy constraint p = q then
26:          if p is not copied locally then
27:             make a local copy p' of p
28:          end if
29:          points-to set(p') $\cup=$ points-to set(q)
30:       end if
31:    end for
32:    $mystate_i$ = Another iteration done
33: end while
```

a static constraint partitioning, which assigns a constraint evaluation to a fixed thread throughout the analysis, achieves only a limited success. On the other extreme, re-calculating the partitions for a perfect load-balance makes the analysis slower than no load-balancing at all! Therefore, we employ a greedy, incremental approach, which achieves an approximately load-balanced threads at a much reduced cost. Our algorithm first distributes copy and load constraints to threads in an even (round-robin) manner, since both kinds of constraints have a singleton write-set (see Table 2). It then makes a single pass over the (costly) store constraints to distribute those to threads again in an even manner. Recall that store constraints may update the points-to sets of multiple pointers. Each thread also maintains a single number indicating its load (amount of work), based on the constraints assigned. In each iteration, as a constraint evaluation results in new points-to information, each thread updates its load-indicator, keeping track of how much information each constraint changed in that iteration. If the new load-indicator is more than its value in the previous iteration by a threshold (pre-determined based on the number of constraints and threads), the thread *orphans* a few (fixed at 5 in our experiments) constraints that added the maximum points-to information and adds those to a shared worklist. Other threads, at the end of each iteration, check this worklist and *adopt* a few constraints if their load-indicator is less than the threshold. Higher value of the threshold results in less number of accesses to the shared worklist with reduced load-balancing, whereas lowering the threshold results in improved load-balancing but more communication (via the worklist) across threads. We experimented with several values for the threshold and the optimal value differs considerably across applications. We found that a threshold set to approximately $7 - 10\%$ of the number of variables achieves a good trade-off for our benchmarks. Note that checking for overload is a (thread-)local strategy, which avoids costly thread-communication overhead. The strategy works reasonably well in practice because the amount of new points-to information added by a constraint in each iteration can be predicted based on that in the previous iteration [22].

4.2 Parallel Online Cycle Elimination

Inclusion-based points-to analysis is generally represented using a constraint graph G wherein a node represents a pointer and a directed edge from node n_1 to node n_2 represents the inclusion relationship points-to set(n_1) $\subseteq$ points-to set(n_2). The points-to information is propagated across the edges. Load and store constraints add more and more edges to G as the analysis progresses generating more opportunities for points-to information propagation. Accumulation of more points-to information at the nodes may result in more edges being added to G. This process is repeated until a fixed-point. Cycles may occur in G at any stage during the analysis. A cycle in G happens due to inter-dependent variables. In terms of RW sets, a cycle indicates a chain $c_1, c_2, ..., c_n$ of constraints such that write-set(c_1) $\cap$ read-set(c_2) $\neq \phi$, write-set(c_2) $\cap$ read-set(c_3) $\neq \phi$, ..., write-set(c_{n-1}) $\cap$ read-set(c_n) $\neq \phi$ and write-set(c_n) $\cap$ read-set(c_1) $\neq \phi$. For instance, a = b and b = a indicates a cycle. An important property of the pointers in a cycle is that all of them (eventually) have the same points-to information. Therefore, to reduce unnecessary propagations, cycles are collapsed. Detecting and collapsing cycles is vital for a scalable inclusion-based points-to analysis [9]. We use Tarjan's algorithm to find strongly connected components (SCC) of a directed graph [29] to detect cycles in the constraint graph.

Cycle elimination involves replacing the nodes in the cycle by a representative node with its points-to information as a union of the points-to information from all the replaced nodes. Further, the incoming edges to and the outgoing edges from the replaced nodes need to be updated to be to and from the representative node respectively. Our algorithm collapses disjoint cycles in parallel. We reuse the same threads as for solving constraints to collapse cycles, since cycle detection and collapsing is done when no threads are solving any constraints.

It is possible for two threads collapsing disjoint cycles to update the incoming edges from the same node, potentially resulting in a conflict. However, this happens infrequently and therefore, we use locking over the nodes to be updated while collapsing cycles.

In order to get maximum benefit out of cycle detection, one needs to carefully tune the cycle detection frequency [12]. We check for cycles once per iteration.

We observed that the advantage of cycle detection is high during only the initial few iterations of the analysis and it gradually reduces as the analysis progresses. Towards the end of the analysis, the cost of cycle detection outweighs its benefits and therefore, we perform cycle detection only upto certain number of initial iterations of the analysis.

4.3 Reducing the Number of Copies

In this section we discuss optimizations that reduce the number of copies of points-to sets across threads. Reducing the number of copies also reduces the number of iterations to reach the fixed-point.

It is unnecessary to make a copy of a variable's points-to set when there is a single writer thread. We detect this situation by maintaining the number of writer threads for each variable and using directly the master copy when no more than one thread writes to the variable. For instance, in the example shown in Table 3, the constraints q = p, r = q, s = r, t = s and *e = a directly update the master copy of the variables in the write-sets.

Our analysis also takes advantage of difference propagation [14] for improving efficiency. Difference propagation involves keeping track of the difference between points-to information of the nodes forming an edge. This helps in propagating only the additional new information across the edge. Our analysis does not initiate a merge operation if the difference between the points-to information of the local copy and the master copy is nil.

While the soundness of our replication-based analysis is oblivious to the way points-to constraints are distributed across threads in each iteration of the analysis, we use a fixed partitioning across all iterations to improve its performance. This helps us cache certain points-to information locally with the thread, updating it using difference propagation with the master only when some other thread has written to it. Using a fixed constraint partitioning also helps each thread make local decisions on the constraint evaluation.

4.4 Limited Scheduling

The amount of new points-to information computed is high in the initial iterations and gradually reduces as the analysis progresses. In fact, towards the end of the analysis, only a few constraints add more points-to information. Therefore, our method restricts parallel analysis to a limited number of iterations. The

Algorithm 3 Context-sensitive analysis.

```
Require: Function f, callchain cc, constraints C, variable set V
 1: for all statements s ∈ f do
 2:    if s is of the form p = alloc() then
 3:       if inrecursion == false then
 4:          V = V ∪ (p, cc)
 5:       end if
 6:    else if s is of the form non-recursive call fnr then
 7:       cc.add(fnr)
 8:       add copy constraints to C for actual and formal arguments
 9:       call Algorithm 3 with parameters fnr, cc, C
10:       add copy constraints to C for return value of fnr and ℓ-value in s
11:       cc.remove()
12:    else if s is of the form recursive call fnr then
13:       inrecursion = true
14:       C-cycle = {}
15:       repeat
16:          for all functions fc ∈ cyclic callchain do
17:             call Algorithm 3 with parameters fc, cc, C-cycle
18:          end for
19:       until no new constraints are added to C-cycle
20:       inrecursion = false
21:       C = C ∪ C-cycle
22:    else if s is an address-of, copy, load, store statement then
23:       c = constraint(s, cc)
24:       C = C ∪ c
25:    end if
26: end for
```

decision of when to change from parallel to sequential analysis is taken based on the amount of new points-to information P_i computed in an iteration i. As soon as it falls below 10% of P_{i-1} computed in iteration $i-1$, our method starts evaluating constraints sequentially.

5 Context-Sensitive Analysis

We extend Algorithm 2 for context-sensitivity using an invocation graph based approach [8]. The approach readily disallows non-realizable interprocedural execution paths. The context-sensitive algorithm starts from function *main* and maintains a stack of function invocations, similar to the runtime. Thus, a *return* from a function always matches the function invocation at the top of the stack. We handle recursion, which can introduce potentially unbounded number of contexts, by iterating over the cyclic call-chain and computing a fixed-point of the points-to tuples. Our analysis is field-insensitive, i.e., we assume that any reference to a field inside a structure is to the whole structure. We do not model `setjmp-longjmp` instructions. Our algorithm handles function pointers similar to [8] by gradually refining the target functions. The context-sensitive version is outlined in recursive Algorithm 3.

The algorithm takes four parameters: the function `f` to be processed, its calling context `cc`, the set of constraints `C` to be generated and the set of variables `V` to be created. The analysis first adds(`g`, {}) to `V` for each global variable `g` where {} denotes an empty context (not shown in the algorithm). It then makes the first call to the algorithm with parameters *main*, {*main*}, C={}, V. The procedure processes all the statements in the function and generates context-sensitive points-to constraints in `C`. `C` is later evaluated using Algorithm 2. Lines 2–5 in Algorithm 3 process memory allocation and create a new variable on encountering an *alloc* statement outside recursion. Lines 6–11 handle a non-recursive call. It first adds the callee to the callchain and then maps the actual arguments to the formal arguments. The algorithm recursively calls itself in Line 9 to process the invocation graph of the callee. The callee is analyzed the same way and the set of constraints `C` keeps getting updated. On the callee function's return, its return value is mapped to the ℓ-value in the call statement. Finally, the calling context is updated by removing the callee. A recursive call is handled in Lines 12–21 by iterating over the cyclic call chain and computing a fixed-point of constraints in `C-cycle`. Note that the recursive call to Algorithm 3 in Line 17 uses the same callchain. The fixed-point over the constraints `C-cycle` generated in the cyclic call graph is then merged with `C` in Line 21. The corresponding context-sensitive constraints for address-of, copy, load and store statements are added in Lines 22–25. A context-sensitive constraint contains variables in a particular context. The two sets, `C` and `V` are finally passed on to Algorithm 2 for solving. The reason for designing the analysis as a two step process (generating constraints and solving them), rather than interleaving the two tasks, is to have a common constraint solving phase (with minor modifications). Thus, Algorithm 2 is used for both context-insensitive and context-sensitive analysis.

Making the analysis context-sensitive increases the number of (context-wise) variables and reduces the sizes of read-write sets for constraints. Thus, making the analysis context-sensitive reduces the number of conflicts across constraints, and in turn, the cost of merging. This helps a context-sensitive analysis achieve a better speed-up over the context-insensitive version.

6 Experimental Evaluation

We evaluate the effectiveness of our approach using 16 SPEC C/C++ benchmarks and five large open source programs, namely *httpd, sendmail, ghostscript, gdb* and *wine-server*. The benchmark characteristics are given in Table 6. All programs are run on an 8-core Intel Xeon E5440 with 2.83 GHz clock, 16 GB RAM running Debian GNU/Linux 5.0.

6.1 Context-insensitive Analysis

Analysis Time. Table 7 shows the speedup obtained using different number of threads. Column SEQ indicates the absolute sequential analysis time in seconds.

Benchmark	KLOC	# Total Inst	# Pointer Inst	# Func
176.gcc	222.185	328,425	119,384	1,829
253.perlbmk	81.442	143,848	52,924	1,067
254.gap	71.367	118,715	39,484	877
255.vortex	67.216	75,458	16,114	963
177.mesa	59.255	96,919	26,076	1,040
186.crafty	20.657	28,743	3,467	136
300.twolf	20.461	49,507	15,820	215
175.vpr	17.731	25,851	6,575	228
252.eon	17.679	126,866	43,617	1,723
188.ammp	13.486	26,199	6,516	211
197.parser	11.394	35,814	11,872	356
164.gzip	8.618	8,434	991	90
256.bzip2	4.650	4,832	759	90
181.mcf	2.414	2,969	1,080	42
183.equake	1.515	3,029	985	40
179.art	1.272	1,977	386	43
httpd	125.877	220,552	104,962	2,339
sendmail	113.264	171,413	57,424	1,005
ghostscript	438.204	906,398	488,998	6,991
gdb	474.591	576,624	362,171	7,127
wine-server	178.592	110,785	66,501	2,105

Table 6: Benchmark characteristics

This base analysis is inclusion-based points-to analysis with offline variable substitution [26] and online cycle elimination [9] implemented. Both the sequential and the parallel implementations use sparse bitmaps to store points-to information. Columns titled 1, 2, 4, 6, 8 indicate the speedup obtained over the sequential version using the said number of threads.

The average speedup as a geometric mean is 3.001 on 8 cores. The best speedup of 4.119 is obtained for perlbmk. Our parallel version takes 10% more time than SEQ for a single thread. However, all the benchmarks we experimented with perform better than SEQ for two (and more) threads.

We compare our results with the parallel points-to analysis built using Galois system [20][1]. Figure 1 shows the average speedups over the base sequential versions. The Galois system obtains an average speedup of 2.314 on the set of benchmarks using 8 cores. Our implementation performs consistently better for any number of threads. It should be emphasized that the parallel points-to analysis in Galois uses speculative parallelism which can rollback an activity if a conflict is detected. Our method uses multiple copies of points-to sets and results in no conflicts across threads. It does not incur any rollback overheads. Therefore, it is suited even for a non-speculative execution. We believe that it is possible to improve Galois speedup by taking advantage of the monotonicity and unordered nature of flow-insensitive points-to analysis.

Memory. Table 7 shows the memory requirements (in MB) of SEQ, our parallel algorithm PARALLEL with 8 cores and the Galois system with 8 cores for each benchmark. On an average, PARALLEL requires 12% more memory than SEQ. This is due to a few additional copies of points-to sets stored locally by

[1] Downloaded from http://users.ices.utexas.edu/∼marioml/hardekopfPointsTo.html.

Benchmark	SEQ	Speedup					Memory (MB)		
	Time(s)	1	2	4	6	8	SEQ	PARALLEL	GALOIS
gcc	6.546	0.80	1.02	2.39	2.96	3.84	83	95	179
perlbmk	2.345	0.92	1.18	2.14	3.03	4.11	100	135	188
vortex	1.445	0.96	1.04	1.97	2.46	3.59	16	24	28
eon	2.446	0.92	1.29	2.41	3.31	4.00	248	314	346
parser	0.889	0.98	1.11	1.74	2.43	3.23	4	7	7
gap	2.777	0.96	1.17	1.69	2.19	2.99	8	13	14
vpr	0.601	0.89	1.18	1.59	2.00	2.46	2	3	3
crafty	0.595	0.99	1.25	1.83	2.38	2.87	1	2	3
mesa	2.045	0.98	1.18	1.77	2.75	3.58	14	16	23
ammp	0.575	0.95	1.03	1.48	1.98	2.68	3	4	5
twolf	0.686	0.97	1.05	1.68	1.99	2.44	4	4	6
gzip	0.456	0.87	1.06	1.72	2.24	2.89	1	1	2
bzip2	0.396	0.90	1.02	1.49	1.80	2.33	1	1	1
mcf	0.382	0.88	1.00	1.36	1.79	2.17	1	1	2
equake	0.436	0.82	1.04	1.47	1.75	2.01	1	1	1
art	0.485	0.84	1.00	1.60	2.10	2.44	1	1	1
httpd	4.447	0.85	1.18	2.13	2.80	3.68	674	705	1028
sendmail	3.311	0.86	1.18	2.07	2.68	3.43	256	279	511
ghostscript	84.497	0.88	1.19	2.22	2.89	3.71	2871	3193	5719
gdb	174.355	0.89	1.09	1.68	2.31	3.01	3556	3976	5362
wine-server	4.452	0.89	1.08	1.78	2.22	2.77	185	210	336
average	14.008	0.91	1.11	1.80	2.35	3.00	382	428	655

Table 7: Context-insensitive analysis: Analysis time, Speedup and Memory

each thread. However, GALOIS requires 53% more memory than PARALLEL and 71% more than SEQ. The authors of GALOIS [20] attribute the high memory consumption to Java implementation. Our C++ implementation performs optimizations to reduce the number of copies of points-to sets across threads.

6.2 Context-sensitive Analysis

Analysis time. The context-sensitive version of our parallel analysis performs similar to its context-insensitive counterpart. The speedup results are detailed in Table 8. In comparison to the context-insensitive results, since variables now have smaller points-to sets, the number of potential conflicts across threads reduces and the analysis requires less number of local copies and merging. Our parallel analysis achieves a speedup of 3.4x on an 8-core machine. Considering that points-to analysis is an *irregular* application with dynamic constraint graph, we believe, this speedup is quite remarkable.

Memory. Memory requirement of our context-sensitive parallel points-to analysis is shown in Table 8. Once again, PARALLEL-CS requires 14% more memory than SEQ-CS.

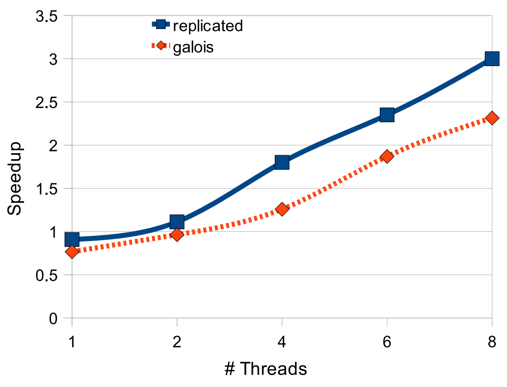

Fig. 1: Speedup comparison between our replication-based approach and Galois system

In summary, our parallel points-to analysis exploits more fine-grained parallelism from programs and promises a scalable approach to parallelizing monotonic, unordered analyses.

7 Related Work

Replication-based Techniques. Replication-based techniques are prevalent in distributed systems, but their main focus is reliability [11, 16]. Bal et al. [2] propose partial replication based techniques for speeding up parallelization. Their approach is based on replicating an object based on its read-write access pattern. While being applicable to parallel points-to analysis, their approach does not exploit the monotonicity property of flow-insensitive analysis, which is key to the arbitrary constraint partitioning employed by our algorithm.

Ziegler et al. [32] propose a data-flow analysis algorithm to uncover the parallelization opportunities for array replication and their temporary privatization. Their algorithm does not take advantage of any monotonic computation.

A few optimistic thread executions involve limited forms of data replication. For instance, in the Grace system [3] used for eliminating concurrency errors, threads execute optimistically and write their updates speculatively but locally. Burckhardt et al. [5] propose isolation types which can be used by threads to read and modify local copies of shared data. Our approach exploits the monotonic and unordered nature of flow-insensitive analysis to eliminate contention.

Prabhu et al. [25] develop an algorithm called EigenCFA for accelerating 0-CFA with a GPU. Similar to our analysis, EigenCFA takes advantage of the monotonicity of 0-CFA to allow stale reads. In their analysis, the same row in the representation matrix may have multiple copies of the same lambda term if they try to add the same information at the same time. They claim that this is a “rare inefficiency”. In contrast, our analysis heavily depends upon this property and exploits it to improve parallelism. Further, the focus of their work is to run

Benchmark	SEQ-CS	Speedup					Memory (MB)	
	Time(s)	1	2	4	6	8	SEQ-CS	PARALLEL-CS
gcc	329.463	0.87	1.12	2.69	3.41	4.28	2859	3419
perlbmk	143.448	0.96	1.29	2.47	3.21	4.03	2133	2628
vortex	91.283	0.98	1.22	2.52	3.22	3.81	1857	2014
eon	93.495	0.96	1.31	2.72	3.41	3.98	1276	1443
parser	35.445	0.99	1.32	2.66	3.33	3.86	478	549
gap	128.478	0.98	1.27	2.50	3.13	3.77	457	514
vpr	29.456	0.93	1.22	2.69	3.08	3.64	735	770
crafty	29.337	0.99	1.28	2.51	2.96	3.46	672	736
mesa	89.388	0.98	1.27	2.01	2.68	3.32	894	949
ammp	34.236	0.96	1.10	1.89	2.77	3.15	427	447
twolf	41.499	0.98	1.10	2.10	2.69	2.90	624	696
gzip	25.234	0.92	1.08	1.74	2.56	2.98	514	641
bzip2	23.322	0.92	1.06	1.85	2.43	2.68	633	686
mcf	22.395	0.91	1.02	1.88	2.60	3.00	403	470
equake	24.306	0.90	1.08	1.96	2.75	3.23	546	610
art	26.459	0.92	1.04	1.92	2.43	2.84	597	656
httpd	224.534	0.89	1.24	2.47	3.01	3.68	991	1131
sendmail	172.743	0.91	1.28	2.58	3.10	3.57	914	1019
ghostscript	4384.238	0.93	1.30	2.66	3.11	3.81	8258	9761
gdb	9338.228	0.93	1.14	2.43	2.99	3.64	5894	6486
wine-server	201.323	0.97	1.16	2.01	2.58	3.10	774	858
average	737.539	0.95	1.19	2.28	2.91	3.44	1521	1737

Table 8: Context-sensitive analysis: Analysis time, Speedup and Memory

the analysis on a GPU whereas our focus is to run it on a multicore.

Sequential Pointer Analysis. The area of sequential points-to analysis is rich in literature. See [15] for a survey. Most scalable algorithms proposed use unification [28][10]. Steensgaard[28] proposed an almost linear time single-pass algorithm that has been shown to scale to millions of lines of programs. However, a unification based approach is very imprecise. Andersen [1] proposed inclusion-based analysis that works on subsumption of points-to sets rather than a bidirectional similarity. An inclusion-based (or subset-based) analysis is more precise than a unification based analysis. However, it is also costly and has a theoretical complexity of $\mathtt{O}(\mathtt{n}^3)$. Several techniques [4][14][30] have been proposed to improve upon the original work by Andersen. [4] extracts similarity across the points-to sets while [30] exploits similarity across the contexts to make use of the Binary Decision Diagrams (BDD) to store information in a succinct manner. The idea of *bootstrapping* [17] first reduces the problem by partitioning the set of pointers into disjoint alias sets using a fast and less precise algorithm (e.g., [28]) and later running more precise analysis on each of the partitions. To address the analysis cost of a completely context-sensitive analysis, approximate representations were introduced to trade off precision for scalability. Das [6] proposed *one level flow*, Lattner et al. [18] unified contexts, while Nasre et al. [23, 21] hashed contexts to alleviate the need to store complete context information.

Parallel Pointer Analysis. In contrast to its sequential counterpart, parallel points-to analysis is still not explored enough. The work on program decomposition identifies various program components on which different analyses can be executed in parallel [31, 27]. Bootstrapping [17] uses partitions of aliases to simulate parallel processing. However, parallelizing is not the main objective of the work and the parallelism extracted is very coarse. Lojo et al. [20, 19] proposed the first parallel implementation of inclusion-based points-to analysis by exploiting the constraint graph formulation. Their analysis works on the assumption of speculative parallelism and an activity may be rolled back if a conflict is detected. In contrast, our work is general and does not require the support of speculative execution. By making multiple copies of points-to sets, our analysis strives to obtain more fine-grained parallelism.

A parallel points-to analysis for object oriented programs is proposed by Edvinsson et al. [7] which deals with different target methods of polymorphic function calls and independent control-flow branches. Maximum speedup obtained has been shown to be less than 2.0 for a set of Java benchmarks. Our parallel analysis takes advantage of monotonicity property of a flow-insensitive analysis to create multiple copies of points-to sets achieving better parallelism.

8 Conclusion

Taking advantage of the multi-core processing requires the analyses themselves to be parallel. While several enhancements have been proposed for sequential pointer analysis, enough work is yet to be done for parallel points-to analysis. By exploiting the monotonicity of flow-insensitive points-to analysis, we proposed a replication-based parallel inclusion-based points-to analysis that extracts more fine-grained parallelism from seemingly sequential programs. We showed the effectiveness of our approach over 16 SPEC 2000 benchmarks and five large open source programs. Our parallel context-insensitive (context-sensitive) analysis achieves a speedup of 3.0x (3.4x) on an 8-core machine and illustrates a promising approach to parallelizing a monotonic, unordered data-flow analysis.

Acknowledgments. We thank Mario Mndez-Lojo for helpful comments.

References

1. Lars Ole Andersen. Program analysis and specialization for the C programming language, PhD Thesis, DIKU, University of Copenhagen, 1994.
2. Henri E. Bal, M. Frans Kaashoek, Andrew S. Tanenbaum, and Jack Jansen. Replication techniques for speeding up parallel applications on distributed systems. Concurrency: Pract. Exper., 4:337–355, August 1992.
3. Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: safe multithreaded programming for c/c++. In OOPSLA, OOPSLA '09, pages 81–96, New York, NY, USA, 2009. ACM.

4. Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, PLDI '03, pages 103–114, New York, NY, USA, 2003. ACM.
5. Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In OOPSLA, OOPSLA '10, pages 691–707, New York, NY, USA, 2010. ACM.
6. Manuvir Das. Unification-based pointer analysis with directional assignments. In Proceedings of the ACM SIGPLAN conference on Programming language design and implementation, PLDI '00, pages 35–46, New York, NY, USA, 2000. ACM.
7. Marcus Edvinsson, Jonas Lundberg, and Welf Löwe. Parallel points-to analysis for multi-core machines. In Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, HiPEAC '11, pages 45–54, New York, NY, USA, 2011. ACM.
8. Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, PLDI '94, pages 242–256, New York, NY, USA, 1994. ACM.
9. Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, PLDI '98, pages 85–96, New York, NY, USA, 1998. ACM.
10. Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, PLDI '00, pages 253–263, New York, NY, USA, 2000. ACM.
11. David K. Gifford. Weighted voting for replicated data. In SOSP, SOSP '79, pages 150–162, New York, NY, USA, 1979. ACM.
12. Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, PLDI '07, pages 290–299, New York, NY, USA, 2007. ACM.
13. Mark Harman, David Binkley, Keith Gallagher, Nicolas Gold, and Jens Krinke. Dependence clusters in source code. ACM Trans. Program. Lang. Syst., 32:1:1–1:33, 2009.
14. Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using cla: a million lines of c code in a second. In Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation, PLDI '01, pages 254–263, New York, NY, USA, 2001. ACM.
15. Michael Hind and Anthony Pioli. Which pointer analysis should i use? In Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis, ISSTA '00, pages 113–123, New York, NY, USA, 2000. ACM.
16. Thomas A. Joseph and Kenneth P. Birman. Low cost management of replicated data in fault-tolerant distributed systems. ACM Trans. Comput. Syst., 4:54–70, February 1986.
17. Vineet Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In Proceedings of the ACM SIGPLAN 2008 conference on Programming language design and implementation, PLDI '08, pages 249–259, New York, NY, USA, 2008. ACM.

18. Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, PLDI '07, pages 278–289, New York, NY, USA, 2007. ACM.
19. Mario Méndez-Lojo, Martin Burtscher, and Keshav Pingali. A gpu implementation of inclusion-based points-to analysis. In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'12, New York, NY, USA, 2012. ACM.
20. Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. Parallel inclusion-based points-to analysis. In Proceedings of the ACM international conference on Object oriented programming systems languages and applications, OOPSLA '10, pages 428–443, New York, NY, USA, 2010. ACM.
21. Rupesh Nasre. Approximating inclusion-based points-to analysis. In Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, MSPC '11, pages 66–73, New York, NY, USA, 2011. ACM.
22. Rupesh Nasre and R. Govindarajan. Prioritizing constraint evaluation for efficient points-to analysis. In Proceedings of the 9th IEEE/ACM international symposium on Code generation and optimization, CGO '11, pages 267 –276, april 2011.
23. Rupesh Nasre, Kaushik Rajan, R. Govindarajan, and Uday P. Khedker. Scalable context-sensitive points-to analysis using multi-dimensional bloom filters. In Proceedings of the 7th Asian Symposium on Programming Languages and Systems, APLAS '09, pages 47–62, Berlin, Heidelberg, 2009. Springer-Verlag.
24. Fernando Magno Quintao Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '09, pages 126–135, Washington, DC, USA, 2009. IEEE Computer Society.
25. Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. Eigencfa: accelerating flow analysis with gpus. In POPL, POPL '11, pages 511–522, New York, NY, USA, 2011. ACM.
26. Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, PLDI '00, pages 47–56, New York, NY, USA, 2000. ACM.
27. Erik Ruf. Partitioning dataflow analyses using types. In Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '97, pages 15–26, New York, NY, USA, 1997. ACM.
28. Bjarne Steensgaard. Points-to analysis in almost linear time. In Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.
29. Robert Tarjan. Depth-first search and linear graph algorithms. SIAM Journal on Computing, 1(2):146–160, 1972.
30. John Whaley and Monica S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In Proceedings of the 9th International Symposium on Static Analysis, SAS '02, pages 180–195, London, UK, 2002. Springer-Verlag.
31. Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: a step toward practical analyses. In Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering, SIGSOFT '96, pages 81–92, New York, NY, USA, 1996. ACM.
32. Heidi E. Ziegler, Priyadarshini L. Malusare, and Pedro C. Diniz. Array replication to increase parallelism in applications mapped to configurable architectures. In LCPC, pages 62–75, 2005.