Inclusion-based pointer analysis using actors

Cosmin Radoi
University of Illinois
cos@illinois.edu

Semih Okur
University of Illinois
okur2@illinois.edu

Abstract

Ubiquitous multicore computers and affordable cloud computation services provides a great opportunity for increasing the speed or precision of program analysis algorithms. We present the first parallel algorithm for a context and field sensitive inclusion-based (Andersen-style) pointer analysis with on-the-fly call graph construction. We show how the pointer analysis and call graph construction can be naturally modeled as an actor system. We implement the algorithm as a drop-in replacement of WALA's (T.J.Watson Libraries for Analysis) pointer analysis and allow for all the flexibility of the original library, i.e., highly configurable context sensitivity, pointer and instance abstraction, etc. We measure the parallel scalability of our algorithm and compare it to the the highly-optimized WALA implementation. We discuss the results, limitations, and avenues for improvement.

1. Introduction

Pointer analysis determines which memory locations are referenced by the pointers in the program. This information is fundamental to a large variety of program verification, optimization, and comprehension techniques [10]. Precise pointer analysis is $\mathcal{NP}$-hard [12] but there are well-researched algorithms that trade precision for better performance [4, 20].

Inclusion-based pointer analysis, introduced by Andersen [4], is on the more precise end of the scale. It computes pointer information by solving a constraint graph with nodes being pointers in the program and edges inclusion relations between the sets of memory locations (i.e., objects in OOP languages) referenced to by the pointers. An initial constraint graph is constructed by parsing the program, with assignments generating inclusion relations and pointer dereferences generating special indirect constraints that can only be solved once the points-to information is known. Thus, pointer information is computed by iteratively solving the constraint graph and adding new inclusion relations from indirect references.

Subsequently, there has been a great deal of work improving the precision of the original algorithm by adding different types of context sensitivity [16, 19] or varying flow sensitivity [11, 14].

Pointer information is also fundamental to computing a precise call graph in languages with dynamic dispatch, as the set of possible target methods can be refined by knowing the possible instances pointed to by the a method's receiver. But pointer analysis can also gain precision and speed from a more precise call graph by not considering the effects of unreachable methods. This mutual dependency led to algorithms that compute the call graph on-the-fly, along with the points-to graph. This approach also allows for context-sensitive call graphs where each node is the abstraction of a method as it is invoked in a specific context.

All these improvements in precision come with a significant performance cost, which has been partially alleviated by a long series of significant enhancement to the original algorithm [6, 7, 18, 21]. Still, flow-insensitive, context-sensitive analyses still fail to scale to programs larger than a few hundred thousands lines of code.

An alternate, less explored, path to better performance is parallelization. The first parallel implementation of an inclusion-based pointer analysis was presented by Méndez-Lojo, Mathew, and Pingali [15]. They express the constraint solving problem as a graph rewrite problem and integrate the offline stage and the Hybrid Cycle Detection technique proposed by Hardekopf and Lin [9]. The solution relies on a precomputed fixed call graph and is context-insensitive.They implemented their system using Galois [17] and achieved a speedup varying between 1x and 3x when compared to an optimized sequential implementation.

We propose an alternate approach to parallelization that takes advantage of the affinity between graph rewriting and actor systems. We express the constraint graph as a graph of actors, with both pointers and instances (memory locations) represented as actors, and set inclusion and dereferencing edges represented by the "knows" relations between actors. Pointer dereferencing is solved by a two-step messaging scheme propagating instance actor addresses from

[Copyright notice will appear here once 'preprint' option is removed.]

$$
\begin{array}{rr}
\text{(variable)} & a, b, c \in \mathbb{V} \\
\text{(field)} & f \in \mathbb{F} \\
\text{(method identifier)} & m, n \in \mathbb{M} \\
\text{(class identifier)} & k \in \mathbb{K}
\end{array}
$$

$$
\begin{aligned}
program &::= program \mid class; program \\
class &::= k' \; extends \; k \; \{ \; methods \; \} \mid k \; \{ \; methods \; \} \\
methods &::= m \; \{s\} \mid methods \; ; m \; \{s\} \\
s &::= a = \text{ new } k \mid a = b \\
&\mid a = b.f \mid a.f = b \\
&\mid a = b_0.m(b_1, b_2, ...) \\
&\mid s; s
\end{aligned}
$$

Figure 1: Language syntax

the dereferenced pointer to the receiving one. The context-sensitive call graph is computed on-the-fly in parallel with the constraint solving.

We implemented the algorithm as a drop-in replacement of WALA's (T.J. Watson Libraries for Analysis) [2] internal sequential pointer analysis. Thus, it allows for all the flexibility present in the original implementation, i.e., n-CFA, n-object sensitivity [16], field-sensitivity, configurable pointer and instance abstractions, etc. The implementation is in Scala and relies on the Akka actor framework [1].

This paper makes the following contributions:

- an actor-model algorithm for computing flow-insensitive, context-sensitive pointer information with on-the-fly call graph refinement. To the best of our knowledge, it is the first actor-model approach to pointer analysis, and the first parallel approach that is both field and context sensitive.
- a reasonably efficient and highly configurable implementation available as a drop-in replacement for the pointer analysis in WALA, a widely used analysis framework
- a preliminary performance evaluation

2. Background

In this section we first present how constraints are generated in an inclusion-based pointer analysis with on-the-fly call graph construction. For presentation purposes, we use the simple language shown in Figure 1 with its intuitive semantics. Still, our implementation does handle the full Java language. The program starts from a predefined object and method.

In an inclusion-based pointer analysis, the instructions in the program generate a set-constraint graph with *base*, *simple*, and *complex* constraints [9**?**]. Method calls generate *interprocedural* constraints which our analysis handles in a context-sensitive manner using a variation of the *lam* constructor [8, 22] (Figure 3).

$$
\begin{array}{rr}
\text{(variable)} & a, b, c \in \mathbb{V} \\
\text{(pointer)} & \Gamma(a) \in \mathbb{P} = 2^{\mathbb{O}} \\
\text{(ordered set of pointers)} & l_x \in List[\mathbb{P}] \\
\text{(statement)} & s \in \mathbb{S} \\
\text{(object)} & o \in \mathbb{O} \\
\text{(object selector)} & o : \mathbb{C} \times \mathbb{S} \to \mathbb{O} \\
\text{(field id)} & f \in \mathbb{F} \\
\text{(field of an object)} & f : \mathbb{O} \times \mathbb{F} \to \mathbb{P} \\
\text{(method identifier)} & m, n \in \mathbb{M} \\
\text{(class identifier)} & k \in \mathbb{K} \\
\text{(context)} & C \in \mathbb{C} \\
\text{(context selector)} & call : \mathbb{C} \times \mathbb{O} \times \mathbb{M} \to \mathbb{C} \\
\text{(instructions for a call)} & instrFor : \mathbb{C} \to \text{"}methods\text{"} \\
\text{(start environment for a call)} & start : \mathbb{C} \to (\mathbb{V} \to \mathbb{P}) \\
\text{(formal parameter)} & param_\Gamma : \mathbb{C} \to List[\mathbb{P}] \\
\text{(formal return)} & return_\Gamma : \mathbb{C} \to \mathbb{P} \\
\text{(local variable environment)} & \Gamma \in \mathbb{C} \times \mathbb{V} \to \mathbb{P} \\
& \Gamma_C \in \{C\} \times \mathbb{V} \to \mathbb{P} \\
\text{(subset)} & l_a \subseteq l_b \equiv \forall i = \overline{0...length(l_a)}. \\
& \quad l_a(i) \subseteq l_b(i)
\end{array}
$$

Figure 2: Domains

The graph's nodes represent pointers and are points-to sets, i.e. sets of abstract objects. The edges are different types of set constraint relations between the pointers. Each variable in the program has an associated pointer and an instantiated object belongs to the points-to set to which it is assigned (*base* constraint). Assignment between two pointers generates a *simple* constraint: the set of the righthand-side pointer is a subset of the lefthand-side pointer – hence the name of this particular pointer analysis.

Reading or writing a field generates a *complex* constraint. When reading field, a subset relation is added between the pointer representing the field of each object referenced by the read pointer and the assigned pointer; similarly for writing a field. This means that complex constraints can only be solved after propagating objects through simple constraints and they, in turn, generate new simple constraints.

Method calls generate *interprocedural* constraints which add inclusion relations between the actual and formal parameters and between method formal return pointers and the as-

$\boxed{C, \Gamma_C \rhd s : \Gamma', K}$

$(base)$ $\quad C, \Gamma_C \rhd a = \text{ new } k : \Gamma_C[a \to \{o(C, \text{"new } k\text{"})\}], \emptyset$

$(simple)$ $\quad C, \Gamma_C \rhd a = b : \Gamma_C, \{\Gamma_C(a) \subseteq \Gamma_C(b)\}$

$(complex)$ $\quad C, \Gamma_C \rhd a = b.f : \Gamma_C, \{\forall o \in \Gamma_C(b) \;\; \Gamma_C(f(o)) \subseteq \Gamma_C(a)\}$
$C, \Gamma_C \rhd a.f = b : \Gamma_C, \{\forall o \in \Gamma_C(a) \;\; \Gamma_C(a) \subseteq \Gamma_C(f(o))\}$

$(interprocedural)$ $\quad \dfrac{o \in \Gamma_C(b_0) \quad C' = call(C, o, m) \quad C', start(C') \rhd instrFor(C') : \Gamma', K \quad b^f = param(C') \quad r^f = return(C')}{C, \Gamma_C \rhd a = b_0.m(b_1, ...) : \Gamma' \cup \Gamma_C, K \cup \{b \subseteq b^f\} \cup \{r^f \subseteq \Gamma(a)\}}$

$(other)$ $\quad \dfrac{C, \Gamma \rhd s_1 : \Gamma', K_1 \quad C, \Gamma \rhd s_2 : \Gamma'', K_2}{C, \Gamma \rhd s_1; s_2 : \Gamma' \cup \Gamma'', K_1 \cup K_2}$

Figure 3: Effects

signed pointers. Contexts are understood in the pointer analysis sense, e.g., whether to clone a method on application is determined by its calling context. We abstract away context selection details in order to allow any kind of calling context sensitivity.

3. Actor-model algorithm

In this section we explain how the constraint generation and constraint solving can be modeled as an actor system. There are two main sources of parallelism that can be exploited:

1. Constraint solving, which has been explored by Méndez-Lojo et al [15]. We give an alternate, actor-model solution.
2. Call graph construction and constraint generation, which is, to the best of our knowledge, novel in both goals and approach.

The system is comprised of the following types of actors (symbol in parenthesis):

- pointer (circle)
- object (square)
- call graph node (pentagon)
- a singleton context selector (octagon)

3.1 Constraint generation

Figure 4 illustrates the constraint generation process. A call graph node represents the evaluation of a method in a particular execution context. The call graph node actor interprets the methods' instructions by creating new actors and sending them messages about their constraints.

An object instantiation generates a points-to (p) relation, i.e. base constraint, between a pointer actor and a newly created abstract object instance actor. An assignment between two pointers generates a subset (s) relation, i.e. a simple constraint, between the righthand-side and the lefthand-side operands. A field read generates a read-field relation, i.e. a type of complex constraint, from the dereferenced object (righthand-side) to the receiving pointer (lefthand-side); similarly for a field write.

In this stage, a method call only informs the actual parameters that they are part of a particular method call in this node. Why this helps and how the actual invocation is realized is discussed in detail in Section 3.3.

3.2 Constraint solving

We solve the set constraint problem by computing a dynamic transitive closure over the constraint graph using actors. Figure 5 illustrates the process that is triggered when a pointer actor b gains a points-to relation to an object o. For each subset-of constraint, b propagates the new points-to constraint to the superset actor, a. When b is the source of a read-field constraint, i.e. a field is read from the pointer, it notifies o that its field needs to be a subset of the assigned pointer, a. When b is the source of a write-field constraint, i.e. a field is written through the pointer, it notifies o which, in turn, lets a know the identity and address of its field, f. Making objects first-class citizens increases the number of messages but it also increases the available parallelism in the system, while also conforming to the actor model restriction of having only immutable shared data.

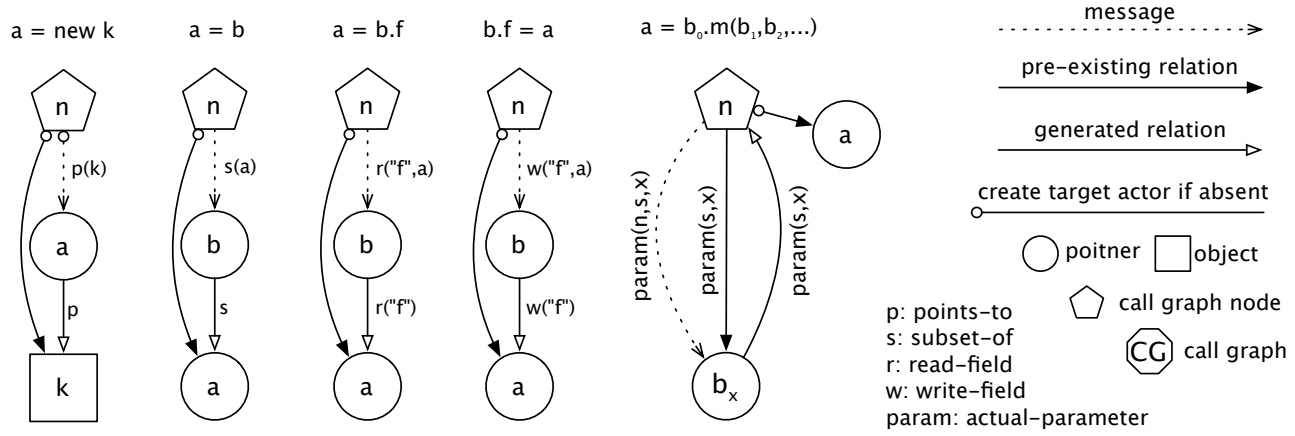

Figure 4: Constraint generation.

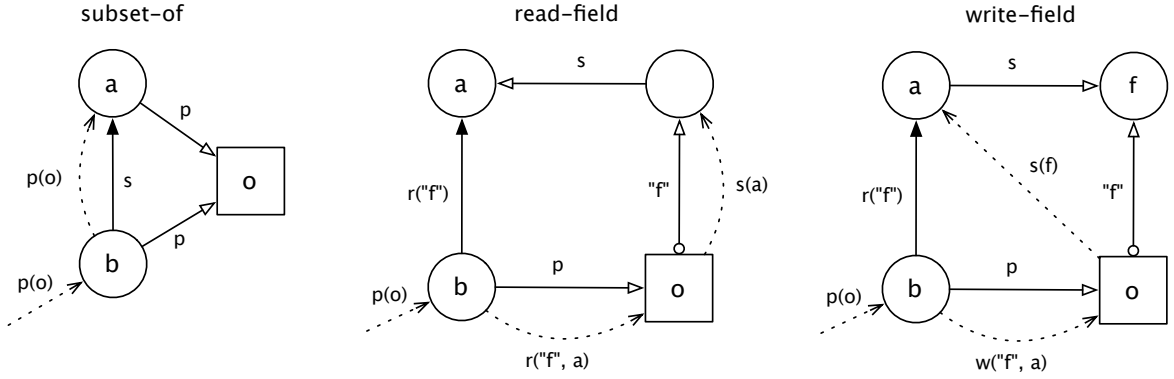

Figure 5: Constraint solving

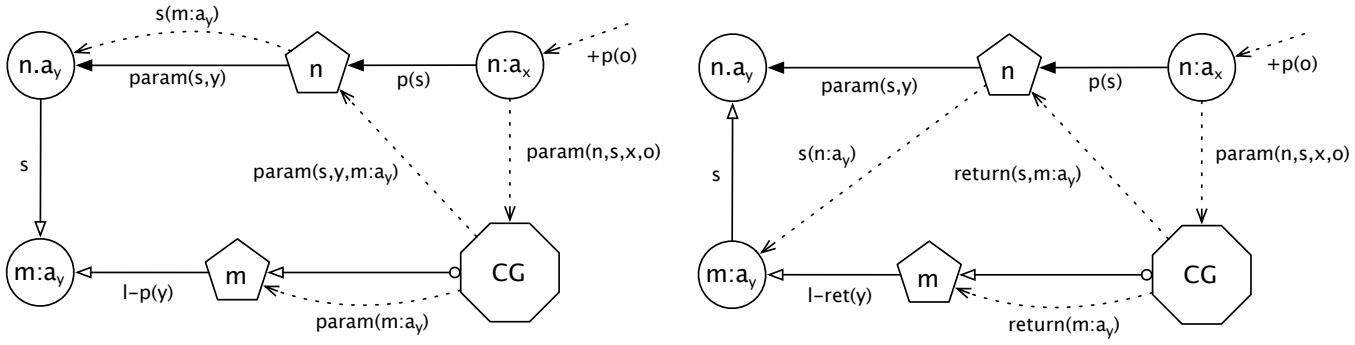

Figure 6: On-the-fly call graph construction. Adding subset constraints from actual to formal parameters (left). Adding constraints from formal return to the assigned variable (right).

3.3 On-the-fly call graph construction

A context sensitive pointer analysis that compute the call graph on the fly *clones* the target method for each distinct discovered context, as illustrated in Figure 6. Each clone interprets all instructions in the method independently, allowing a great amount of parallelism.

What constitutes a distinct context can be configured by specifying a *call* method (see Fig. 2). This *call* method takes as arguments the current context, the receiver object, and the called method. The context distinctiveness is a global property so parallelizing this process involves either performing redundant computation or having a single decision actor. We choose the latter route by creating a specialized actor, the ”CS“ (context selector) actor, that has the sole purpose of identifying and recording unique contexts.

Knowing the pointer to a method’s actual parameters is not enough for determining its context. The context is directly dependent on the receiver object, both for determining the target of dynamic dispatch and for object-sensitive analyses [16]. But the receiver abstract object (or objects) may only be available after constraint solving. We solved this problem by not trying to determine the target method and context when encountering a method call, but delay it to the pointer the required information is available. Thus, on interpreting a method call, the call graph node actor only notifies the receiver pointer that it is part of a particular method invocation in this particular node. Then, when the pointer discovers a new pointer object, it notifies the call graph actor. The call graph actor then determines whether the new object generates a new distinct context and, if so, spawns a new call graph node. Also, it sends the caller call graph node the pointers for the formal parameters and return pointers of the callee.

4. Evaluation

The primary purpose of the evaluation is to determine how fast and scalable our approach is. We measure the runtime of our implementation when analyzing a set of nine Java applications. We then analyze the same applications using WALA[2] , a state of the art, highly optimized pointer analysis. Finally, we compare and discuss the results.

We implemented our analysis in Scala using the Akka actor framework. We evaluate using a machine with 4 Intel Xeon E7-4860 processors and 132GB of RAM. Each Intel Xeon E7-4860 has 10 cores and hyper-threading, thus the machine totals 40 cores with 80 physical threads.

We evaluate on nine Java applications from the DaCapo 9.12 benchmark suite [5]. The applications, shown in Table 1[1], range from 1k to 300k source lines of code. For each application, we run the pointer analysis with varying levels of precision, between 0-CFA and 2-CFA.

[1] description from [5]

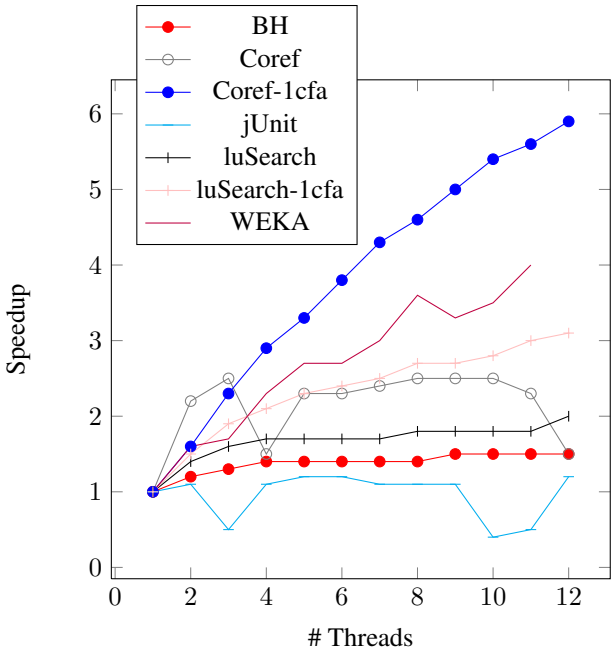

Figure 7: Scalability of our original prototype

Problem size Better precision increases the problem size but the magnitude of this increase is not easily predictable as it is highly dependent on the shape of the call graph. E.g., if a method is used in 100 places in a program and we increase the precision from 0-CFA to 1-CFA, there will be 99 new call graph nodes, each also adding more nodes to the heap graph. If the same method appears only once, the precision increase will not cause an increase in the problem size. So, the potential increase in problem size is exponential but it is not easily predictable. In practice, it is usually under one order of magnitude increase for each precision step (see the # CG nodes column in Table 2).

Results Table 2 shows WALA’s runtime along with APA’s runtime when using only one thread (seq), and using the optimal number of threads (best parallel).

Figure 8 shows, for each project under varying precision, the speedup obtained when varying the number of threads made available to our algorithm. We limited the runtime to 10 minutes so the graph is missing some values due to timeouts. In particular, only avrora finished for 2-CFA within the allocated time. Additionally, tradebeans did not finish for 1-CFA when using more than 32 threads, and fop only finished for 1-CFA when using 8 and 16 threads so we could not compute a speedup value.

Performance Unfortunately, the results of the best version of our algorithm are below expectations both in terms of speed and scalability. We were initially excited about our algorithm as an early prototype implementation showed good scalability. Figure 7 shows the results of a preliminary evaluation of an early version of our algorithm. While $2-6\times$

Project	Description
avrora	avrora simulates a number of programs run on a grid of AVR microcontrollers
pmd	analyzes a set of Java classes for a range of source code problems
sunflow	renders a set of images using ray tracing
luindex	Uses lucene to indexes a set of documents
tradebeans	runs the daytrader benchmark with an in memory h2 as the underlying database
h2	executes a number of transactions against a model of a banking application
fop	takes an XSL-FO file, parses it and formats it, generating a PDF file.
tomcat	runs a set of queries against a Tomcat server retrieving and verifying the resulting webpages
lusearch	Uses lucene to do a text search of keywords over a corpus of data

Table 1: Evaluation suite.

			WALA	APA (our)	APA (our) best parallel		
Project	Precision	# CG nodes	t(s)	1-thread t(s)	total t(s)	seq t(s)	# threads
avrora	0-CFA	1399	4	11	10	1	8
	1-CFA	4358	5	11	11	2	1
	2-CFA	8149	7	12	11	2	4
pmd	0-CFA	6277	24	55	33	6	4
	1-CFA	37290	50	407	234	77	8
sunflow	0-CFA	3667	6	17	12	3	4
	1-CFA	14430	13	61	29	4	4
luindex	0-CFA	3452	9	18	17	3	2
	1-CFA	17146	14	78	35	6	8
tradebeans	0-CFA	1072	3	11	5	1	4
	1-CFA	2938	4	11	6	2	4
h2	0-CFA	2093	5	11	11	2	1
	1-CFA	8952	11	48	21	3	8
fop	0-CFA	6192	29	325	210	64	8
tomcat	0-CFA	4734	14	40	25	4	8
	1-CFA	25889	22	132	65	19	8
lusearch	0-CFA	3588	8	17	11	2	4
	1-CFA	18161	14	71	33	6	8

Table 2: Results. The number of call graph nodes is as reported by our tool. There is up to 10% difference between our tool and WALA in the number of discovered call graph nodes. The difference is due to subtle differences in the implementation, e.g. the handling of library methods. The fifth column shows the runtime of our algorithm when run on a single thread. The sixth column shows the runtime of our algorithm with the optimal number of threads. The seventh column shows the time spent by our algorithm in a sequential section. # threads is the minimum number of threads required for the best parallel time result.

speedup on 12 cores might not seem like much, it is very good in the context of pointer analysis, which is a hard case of irregular parallelism[17], as the best competing algorithm scales up to $3\times$ [15]. Still, our implementation was more than one order of magnitude slower than WALA so the initial results were not conclusive and we need to improve speed.

5. Discussion

Set implementation The most computationally intensive parts of a pointer analysis are the set difference and set union operations in constraint solving. Our initial, naive implementation used hash sets. After experimenting with alternative set implementations, we found sorted word-aligned compressed bitmaps[13] to be the best performing. The use of integer arrays requires a central repository that assigns indexes to objects but the extra communication is not very expensive. Using compressed bitmaps decreased the runtime of our algorithm by $2 - 10\times$.

WALA uses a highly-efficient bitmap implementation that employs a *shared* repository of canonical sets. We experimented with using WALA's bitmaps with *synchronized* shared repository accesses and found that, despite high lock contention, the performance was close to using the lock-free word-aligned compressed bitmaps. We believe that, aside from lack of communication, the highly efficient *shared-*

memory bitmap implementation is the reason WALA has better performance than our tool.

Improving improving set performance does make our algorithm faster but, as the relative cost of the sequential section is increased, it also makes the algorithm less scalable, as Figure 8 shows.

Available parallelism The seventh column in Table 2 shows the time spent in the sequential part of our algorithm. Most of this time is spent gathering the results from the actor system into the main thread. The sequential work is between 5 and 20 percent of the total work.

As the computation is irregular, it is hard to measure the available parallelism during the parallel computation. As a proxy, we profiled the analysis' CPU usage. The analysis tends to fully use all the threads we make available while in the parallel section. The thread utilization only drops during garbage collection and at the very end of the algorithm, while the actor system is shutting down. This shutting down period takes at least 5 second so it is responsible for the poor scalability of the analysis on small problems.

The 5 to 20 percent sequential section, combined with the apparently very good available parallelism in the parallel section, should make the algorithm fairly scalable. Still, the speedup we observe is much less than what Amdahl's law predicts. We believe this is due to poorly scalable communication.

Communication Actors communicate by exchanging messages and Akka has a very performant implementation. In one experiment, the Akka team shows how it scales to 20 million messages per second with 128 actors on an 48 core AMD Opteron [3].

Still, the number of messages used by our algorithm, i.e. several millions for a medium-size problem, does take its toll. By profiling, we found that the communication cost is roughly between one quarter and one half of the overall computation. The estimation is rough because precise profiling of Java programs is still problematic.

We attempted to reduce the cost of communication by *bundling* messages between actors. When an actor needs to send several messages to another actor as part of the same task, we bundle them together and unpack them on the other side. This improved performance but not significantly.

Context switching We varied the number of threads between 1 and 256. As the evaluation machine has 4 processors, each with 10 hardware cores with hyper-threading, it is interesting to observe how context switching and cache line invalidation affects performance. Figure 8 highlights the thread values 10 (all hardware cores on one processor), 20 (all hardware threads on one processor), 40 (all hardware cores), and 80 (all hardware threads). While we haven't evaluated for these particular values, we can observe how runtime evolves between 8, 16, 32, and 64 threads.

We observe that hardware architecture does have a major impact on scalability. For most benchmarks, the best results are obtained when using 8 or 16 threads, and adding more threads makes the performance degrade significantly. This is most likely due to the higher cost of communicating between different processors. This also suggests that the algorithm would not distribute well.

6. Conclusion

We propose a novel, actor-based algorithm for pointer analysis. The algorithm shows some level of scalability, up to 2.3x when using 8 threads, and is fairly fast, but it fails to beat the highly optimized WALA implementation. Our scalability result is in line with other work in this area [15]. Mendez et al.'s [15] obtained up to $3\times$ speedup versus their parallel implementation when using one thread, and up to $2\times$ speedup versus their own implementation of a state-of-the-art sequential algorithm.

While our parallel implementation is not better than a highly-optimized sequential one at this point, the techniques we propose could be useful in developing a better parallel algorithm. Also, improvements in hardware, e.g. transactional synchronization extensions, and in the Akka framework might tip the scale in favor of our algorithm.

References

[1] Akka actor framework. URL `http://akka.io/`.

[2] Wala documentation. URL `http://wala.sourceforge.net/`.

[3] 2 2012. URL `http://letitcrash.com/post/17607272336/scalability-of-fork-join-pool`.

[4] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, Oct. 2006. ACM Press. doi: http://doi.acm.org/10.1145/1167473.1167488.

[6] M. Fähndrich, J. S. Foster, Z. Su, and A. 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. ISBN 0-89791-987-4. doi: 10.1145/277650.277667. URL `http://doi.acm.org/10.1145/277650.277667`.

[7] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM SIGPLAN 2000 conference on Pro-*

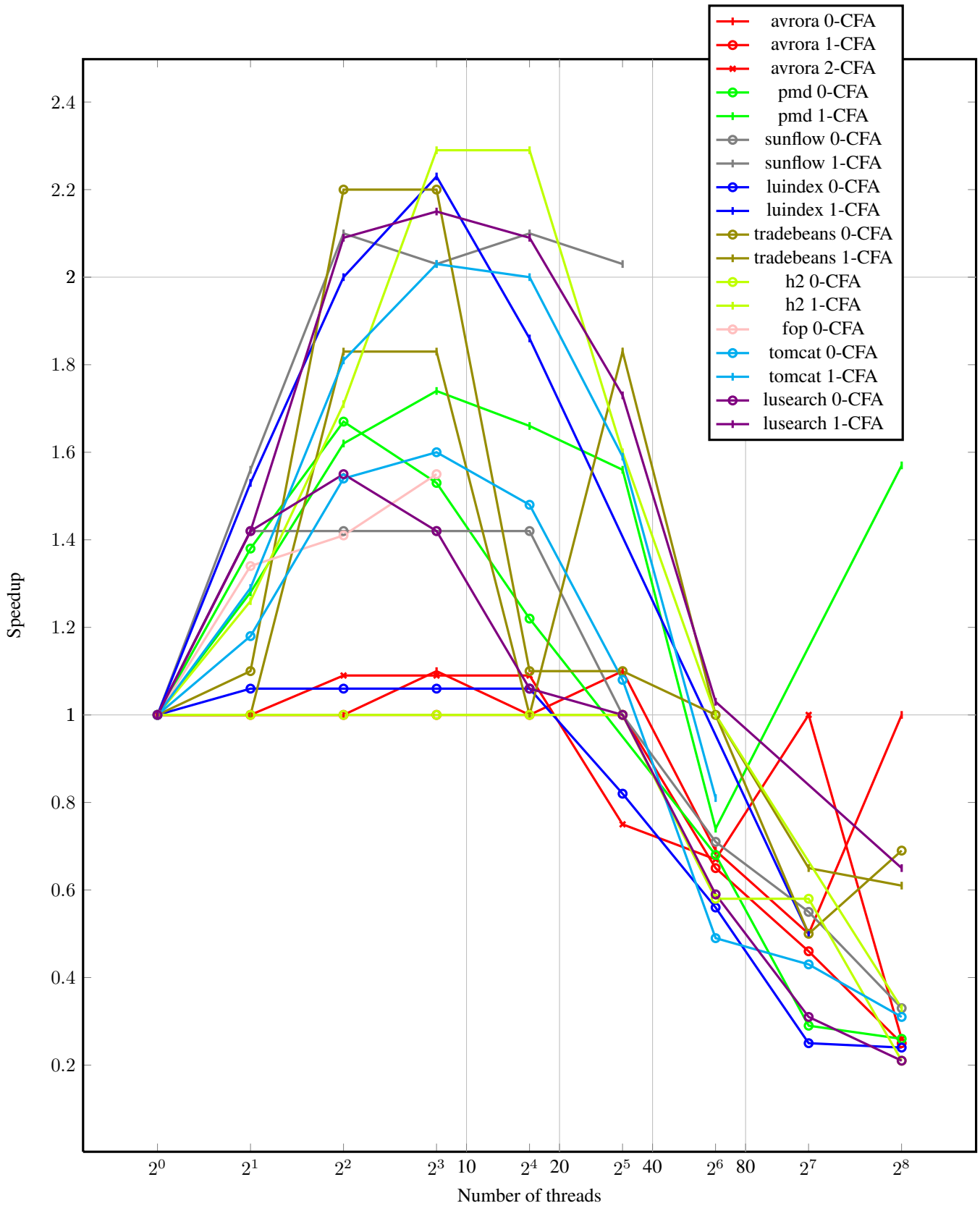

Figure 8: Results

gramming language design and implementation, PLDI '00, pages 253–263, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349332. URL http://doi.acm.org/10.1145/349299.349332.

[8] J. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for c. *Static Analysis*, pages 157–171, 2000.

[9] B. Hardekopf and C. 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. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250767. URL http://doi.acm.org/10.1145/1250734.1250767.

[10] M. Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM. ISBN 1-58113-413-4. doi: http://doi.acm.org/10.1145/379605.379665. URL http://doi.acm.org/10.1145/379605.379665.

[11] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, July 1999. ISSN 0164-0925. doi: 10.1145/325478.325519. URL http://doi.acm.org/10.1145/325478.325519.

[12] W. Landi and B. G. Ryder. Pointer-induced aliasing: a problem taxonomy. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '91, pages 93–103, New York, NY, USA, 1991. ACM. ISBN 0-89791-419-8. doi: 10.1145/99583.99599. URL http://doi.acm.org/10.1145/99583.99599.

[13] D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *Data & Knowledge Engineering*, 69 (1):3–28, 2010.

[14] O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 3–16, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: http://doi.acm.org/10.1145/1926385.1926389. URL http://doi.acm.org/10.1145/1926385.1926389.

[15] M. Méndez-Lojo, A. Mathew, and K. Pingali. Parallel inclusion-based points-to analysis. In *OOPSLA*, OOPSLA '10, pages 428–443, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6. doi: http://doi.acm.org/10.1145/1869459.1869495. URL http://doi.acm.org/10.1145/1869459.1869495.

[16] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14:1–41, January 2005. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/1044834.1044835. URL http://doi.acm.org/10.1145/1044834.1044835.

[17] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993501. URL http://doi.acm.org/10.1145/1993498.1993501.

[18] A. Rountev and S. 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. ISBN 1-58113-199-2. doi: 10.1145/349299.349310. URL http://doi.acm.org/10.1145/349299.349310.

[19] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[20] B. 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. ISBN 0-89791-769-3. doi: 10.1145/237721.237727. URL http://doi.acm.org/10.1145/237721.237727.

[21] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: reducing redundancies in inclusion constraint graphs. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 81–95, New York, NY, USA, 2000. ACM. ISBN 1-58113-125-9. doi: 10.1145/325694.325706. URL http://doi.acm.org/10.1145/325694.325706.

[22] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 1–12, New York, NY, USA, 1995. ACM. ISBN 0-89791-697-2. doi: 10.1145/207110.207111. URL http://doi.acm.org/10.1145/207110.207111.