Parallel Points-to Analysis for Multi-Core Machines

Marcus Edvinsson
marcus.edvinsson@lnu.se
Jonas Lundberg
jonas.lundberg@lnu.se
Welf Löwe
welf.lowe@lnu.se
School of Computer Science, Physics and Mathematics
Linnaeus University
35195 Växjö, Sweden

ABSTRACT

Static program analysis supporting software development is often part of edit-compile-cycles, and precise program analysis is time consuming. Points-to analysis is a data-flow-based static program analysis used to find object references in programs. Its applications include test case generation, compiler optimizations and program understanding, and more. Recent increases in processing power of desktop computers comes mainly from multiple cores. Parallel algorithms are vital for simultaneous use of multiple cores. An efficient parallel points-to analysis requires sufficient work for each processing unit.

The present paper presents a parallelized points-to analysis of object-oriented programs. It exploits that (1) different target methods of polymorphic calls and (2) independent control-flow branches can be analyzed in parallel. Carefully selected thresholds guarantee that each parallel thread has sufficient work to do and that only little work is redundant with other threads. Our experiments show that this approach achieves a maximum speed-up of 4.43 on 8 cores for a benchmark suite of Java programs.

Categories and Subject Descriptors
C.1.2 [PROCESSOR ARCHITECTURES]: Multiple Data Stream Architectures (Multiprocessors)—Parallel processors; D.1.3 [PROGRAMMING TECHNIQUES]: Concurrent Programming—Parallel programming

General Terms
ALGORITHMS, PERFORMANCE, EXPERIMENTATION

Keywords
Data flow analysis, Parallel processing, Parallel algorithms, Program analysis

1. INTRODUCTION

Static program analysis is used in tools supporting software development and optimization. Program analysis supporting software development is integrated in edit-compile-cycles and sufficiently accurate responses should be provided quickly. This requires an efficient computation of program analysis. Program analysis supporting software optimization is less critical regarding response time but benefits from high accuracy; as analysis time grows exponentially with the accuracy required, even this application domain would benefit from efficient computations.

Points-to analysis is a static program analysis based on data-flow algorithms that extracts reference information from a given input program, i.e., possible targets of a call and possible objects referenced by fields. Examples of applications using points-to analysis results are: metrics analyses computing coupling and cohesion between objects [4, 12] and architectural recovery by class clustering proposing groupings of classes, either based on coupling and cohesion or directly on reference information [28, 25]. Source code browsers compute forward and backward slices [13] of a program point which, in turn, requires reference information. In software testing, class dependencies determine the test order [3, 30, 22]. Reverse engineering of UML interaction diagrams requires very precise reference information in order to be useful [31]. Finally, static design pattern detection needs to identify the interaction among participating classes and object instances in order to exclude false positives [26].

The traditional approach to increase the computational power of processors was to fit more transistors on a chip and increase their clock speed, thus getting more powerful processors [11]. The increased switching of transistors generate more heat and consume more power. It is possible to mitigate these problems to some extent. Instead, single processor chip contains multiple cores [6], each core with at least the same processing power of previous single-cored processors. Major processor suppliers have released processors with 2, 4 and 6 cores and are currently working on processors with 8
and 16 cores [21, 11]. Moreover, even standard desktop computers can contain more than one such processor. Applications can take advantage of processing power from multiple cores if (and only if) independent application tasks are processed in parallel, e.g., using multiple cores allocating each independent task on a separate core. However, this requires that the solution applications define independent tasks for the problems they solve.

The traditional approach to data-flow analysis, in general and points-to analysis specifically, as used in today’s Integrated Development Environments and optimizing compilers is sequential and, therefore, the increased computational power of commodity processors would not lead to a significant speed-up of program analysis.

Points-to analysis is time consuming. Finding a solution, i.e., calculating the analysis results, requires fix-point iterations over analysis computations for individual program entities, so-called transfer functions, and propagations of their results along control- and data-flow dependencies between the program entities. For a medium sized program of 1 million lines of code, we speak of about 5 million individual program entities whose transfer functions need to be updated several times until a fixed point is reached.

In principle, the parallelization of points-to analysis is trivial since theoretically any fair order of updating the transfer functions will eventually stabilize to a fixed point. Hence, even any fair parallel update order will do. However, the orderings differ in the number of updates necessary to reach the fixed point and, hence, in the work of the analysis program. Theory and heuristics to reduce this work advise us to: update the transfer functions along data and control dependencies in a data-driven manner, and stabilize inner loops before outer loops. This restricts the naïve parallel solution quite a bit, but it still leaves some degrees of parallelism. Branches in a program, due to conditional statements and polymorphic calls, could be analyzed in parallel. Corresponding attempts with modest success date back to the early 1990s, when parallel processing was dominated by distributed memory machines and, hence, only coarse-grained parallelism paid off (cf. Section 6).

For all benchmark programs in their experimental evaluation, an analysis of the inherent parallelism of the parallelized program analysis problem promised a potential speed-up scaling well to up to 16 processors [10]. However, a first implementation of parallelized program analysis [9], only using parallelism induced by polymorphic calls, did not reach these speed-ups. Only five out of eight benchmarks showed a speed-up on an 8-core architecture and the average speed-up over all benchmarks was well below 2.

The current paper presents a parallelized points-to analysis which goes beyond the aforementioned results. The analysis (i) exploits polymorphic calls and control-flow branches for the parallelization. Using the same benchmark suite as [9], the analysis implementation (ii) shows a speed-up of points-to analysis on all benchmark programs on an 8-core architecture. Even here, the speed-ups observed are better than the inherent parallelism of the problems suggests, which the paper (iii) analyzes and explains with reduced work in the parallel executions.

The challenges with analysis parallelization are finding independent tasks and efficiently executing them concurrently. When tasks are identified they are mapped to threads. Control flow join points require synchronization after all tasks analyzing independent branches are completed. It is necessary to control which tasks spawn threads, since an overhead is associated with thread management and synchronization.

The remainder of this paper explores how this can be done, thus organized as follows: Section 2 introduces the foundations of program analysis in general and points-to analysis specifically. Section 3 presents three opportunities to parallelize the analysis algorithm. Section 4 presents four variants to performing parallelized points-to analysis. Section 5 presents the evaluation of these variants including benchmarks, metrics, and results. Section 6 discusses related work. Section 7 concludes the paper and suggests directions for future work.

2. POINTS-TO ANALYSIS

The basis for points-to analysis, and program analysis in general, are the theories of monotone dataflow frameworks [20, 24] and abstract interpretation [5, 24]. A program is represented by a program graph; its nodes correspond to program points, its edges to control and data dependencies between them. The analysis iteratively computes values for each node by merging values from predecessor nodes and by applying transfer functions representing the abstract program behavior at these nodes.

Our approach to points-to analysis has been presented in detail before [19, 18]. We include a brief summary here for completeness.

2.1 Program Representation

A points-to analysis computes sets of references to abstract objects. An abstract object \( o \) is an analysis abstraction that represents a set of run-time objects. In our case: all run-time objects created at the same syntactic creation point \( s \) correspond to the same unique abstract object \( o_s \).

In the analysis, reference variables will in general hold references to more than one abstract object. Hence, we assume that each points-to value \( v \) in the analysis of a program is an element in the points-to value lattice \( L_V = \{V, \cup, \cap, 0, \top \} \) where \( V = 2^V \) is the power set of \( O, \top = O, \bot = \emptyset \), and \( \cup, \cap \) are the set operations (union) and (intersection). We use the notation \( Pt(a) \) to refer to the points-to value that is referenced by the expression \( a \).

Our points-to analysis uses an SSA-based program representation [7, 23] where each method in the program is represented by a method graph. Nodes correspond to operations and local variables are resolved to data-flow edges connecting the unique operations (nodes) that define a variable to operations (nodes) that use the vari-
able. As a result, every def-use relation via local variables is explicitly represented as an edge between the defining and using operation. Join-points in the control flow where several definitions may apply are modeled with special $\phi$-operation nodes.

The method graphs can be seen as semantic abstractions of methods, an SSA graph representation specially designed for points-to-analysis. It is an abstraction since we have removed all operations not directly related to reference computations, e.g., operations related to primitive types. Moreover, we abstracted from the semantics of the remaining operations by giving them abstract analysis semantics.

2.2 Simulated Execution

The node updating strategy we use is simulated execution, described more in detail in [19]. It simulates program execution: the nodes of a method are analyzed in a topological order. Inner loops are stabilized before outer loops. When call nodes are analyzed (monomorphic call nodes $\text{MCall}^m$ with target method $m$, polymorphic call nodes $\text{PCall}^{tgts}$ with a set of target methods $tgts$), the currently analyzed method is put on hold, pushed onto a call stack, and analysis of the target method(s) $m$ starts. The simulated execution analysis is recursively applied to the target(s) $m$. When the analysis of $m$ has stabilized, simulated execution resumes with the $\text{caller}$, which is popped from the call stack. Recursive calls in analyzed programs would lead to non-terminating loops. When a recursive call is found, i.e., a target method $m$ is already on the call stack, then $m$ is marked as recursive and the bottom (empty) analysis results are assumed to be returned from $m$. If a method $m$ is marked recursive after return to a $\text{caller}$, it is reanalyzed immediately until a fixed point is reached.

2.3 Call Graphs and Reachability

The order in which the methods of a program are analyzed is determined by the program’s call graph. A call graph is a directed graph with methods as nodes and edges ($\text{caller}, m$) if method $\text{caller}$ contains an $\text{MCall}^m$ or a $\text{PCall}^{tgts}$ SSA node with $m \in tgts$. The reachability set of a method $\text{caller}$ is the set of methods that are transitively connected from $\text{caller}$ in the call graph.

Call graph and reachability set, resp., can be computed statically such that they are over-approximations of the actual calls and the actual reachable methods, resp., under any program execution.

3. PARALLELIZED SIMULATED EXECUTION

Many SSA nodes in a Points-to SSA graph are sequently dependent and their parallel execution does not lead to any speed-up, just to thread-switch and synchronization overhead. Therefore, sequently dependent nodes ought to be assigned to a single thread.

There are three constructs which lead to independent nodes in the program to analyze:

(a) Selection branches,

(b) Polymorphic calls, and

(c) Certain context definitions in context-sensitive analysis where a called method is independently analyzed for each of the distinct target contexts (not detailed here).

The main task of defining a parallelized simulated execution is to assign clusters of SSA nodes, i.e., sets of SSA nodes, of a program to individual threads. Parallel threads ought to get assigned independent nodes: $m_1$ and $m_2$ are independent methods if they are not reachable from one another in a call graph, and $n_1$ and $n_2$ are independent SSA nodes if they are (i) not reachable from one another in an SSA graph of a method, and (ii) the methods they are contained in are not reachable from one another either. Dependencies between nodes of different clusters, hence threads, are discouraged, since they induce synchronization and reduce parallelism.

In fact, our goal is to find a clustering of SSA nodes such that the induced processing system stabilizes in a fixed point and terminates in minimum time. For a number of reasons, we must not expect to find an optimal solution. First, it is not decidable before the actual analysis how many iterations over loops and recursive parts of the program are necessary in order to compute the fixed point. Second, even in the absence of iteration and recursion, our problem can be reduced from a general task graph clustering problem which is known to be NP-hard even for very simple cost models. Hence, we will apply heuristics.

Clusters may not be fully node-disjoint sets. It is known from task graph clustering that such redundant clustering may reduce the overall makespan. Our heuristics allow that a node is assigned to different clusters. However, redundant clustering also increases the overall work and implicitly introduces dependencies over shared result variables, i.e., the analysis result of the shared nodes, for the nodes writing values, and for the side effects in the data structure modeling the heap memory.

3.1 Independent Branches (a)

Each selection branching in a method’s SSA graph has two independent sets of nodes, i.e., the set of nodes in the true branch and the set of nodes in the false branch. As long as disjoint sets of methods are called, the methods reachable via call nodes in the two selection branches are disjoint as well. In general, however, this cannot be guaranteed, since the same target methods may be reachable (transitively called) from the two selection branches. Our heuristic will in principle allow this source of redundancy; our cost model will, however, reduce it.

3.2 Polymorphic Calls (b)

A polymorphic call (PCall) targets a number of methods, defined in the declared type of the designated expression or in sub-, or super classes thereof. Simulated execution may process all (potential) call targets in parallel. As observed under (a), assigning the reachable nodes of the different target methods in a cluster each may lead to redundant clusters, since some common
methods may still be reachable (transitively called) from the different call targets.

4. IMPLEMENTATIONS

This section discusses parallelized analysis variants using the simulated execution node updating strategy based on (a) independent branches and (b) polymorphic calls.

4.1 Baseline

This parallelization of analysis eagerly executes all independent tasks in parallel. A task is created for each branch and for each polymorphic target method that should be analyzed independently. The reachability of a task is defined as either the union of reachability of all methods called from a branch or the reachability of the target method of a polymorphic call. Instead of naïvely creating a new thread for each task, we use a fixed-size pool of threads. The pool of threads is created upon initialization; its optimum size depends on the number of processors at hand. No more than these initial threads will be created during the simulated execution.

Tasks are scheduled to threads. Therefore, tasks are put in a queue after creation. The threads iterate over the following two simple steps: (i) get task from queue, (ii) execute task. The queue is blocking, which leaves threads waiting if the queue is empty. When a thread has finished the execution of a task, it tries to take a task from the queue and waits if the queue is empty or until tasks are added to the queue again.

Example 1 The algorithm for analyzing polymorphic calls is listed in Figure 1. The algorithm iterates over the target methods of a polymorphic call (line 20). For each target method, it creates a new task and stores it in the task set tasks (lines 21 and 22). The executeAndWait() call, line 24, asks the thread pool pool to eventually execute the tasks. Threads that are waiting to take tasks from the queue are resumed, and one of them processes the task. Note that the successor node of a PCall node is not executed before all tasks are processed.

4.2 Threshold

The overhead of parallel execution includes task creating, task queue and thread pool handling, synchronization, blocking, and resuming threads. It will not pay off if the tasks are too lightweight. Therefore, two thresholds $T_p$ and $T_b$ are added to the baseline approach, deciding which tasks are worth being processed independently.

The Threshold approach uses a simple cost model here: the workload of a task is the number of methods in its reachability set. The motivation for this cost model comes from the lower bound of updating all transitively called methods’ nodes at least once. It ignores the number of nodes in a method and the variations in time needed for updating these nodes, and it is unaware of the actual number of iterations needed to reach a fixed point. A new task is created if the size of its reachability set is larger or equal to a threshold value. Two thresholds are used, $T_p$ for polymorphic methods and $T_b$ for branches. Otherwise, it is executed sequentially by the current thread. The Baseline approach corresponds to the Threshold approach with $T_p = 1$ and $T_b = 0$.

Compared to Baseline, Threshold reduces the parallelism but, it avoids creating and executing too small tasks which do not justify the overhead of parallel execution. Appropriate values for $T_p$ and $T_b$ are determined experimentally, cf. Section 5.

4.3 Avoid Redundancy

This approach is an extension of Threshold. It avoids overlapping reachability sets between methods of independent tasks. Redundancy between two tasks makes it more likely that synchronization is required at runtime and that the parallelism is reduced due to corresponding overheads and serializations.

At runtime, the approach inspects (i) all tasks that are currently executed and (ii) all tasks that are waiting to be executed. A new task (and eventually a new thread) is only created for a branch $b$ or polymorphic target method $m$ if its reachability set is disjoint with the concerned tasks’ reachability sets. The work for each task is controlled with the thresholds $T_p$ and $T_b$ are also used in this variant.

A decision table for checking the redundancy – used to support the dynamic decision of parallelizing or not – is created statically before the analysis is started.

4.4 Reduce Redundancy

The total avoidance of redundancy is quite a strong restriction. Experiments showed that the number of created tasks is largely reduced and the analysis time is generally affected negatively compared to the Threshold approach. To help this, the Reduce Redundancy approach implements a variant of Avoid Redundancy. It allows overlap between tasks.

Figure 1: Algorithm for creating and executing tasks independently.
Controlling the redundancy works similar to controlling the workload in the Threshold approach. A new threshold value \( T_\ell \) for the largest allowed redundancy is introduced. If and only if a potentially independent task has a smaller accumulated redundancy value with the inspected (running or waiting) tasks, it will be created and then executed independently. The work for each task is controlled with \( T_p \) and \( T_b \). As for \( T_p \) and \( T_b \), the appropriate threshold value of \( T_\ell \) is determined experimentally.

### 4.5 Algorithmic Generality

Lee et al. [16] identify three different types of parallelism in data-flow algorithms: independent-problem, separate-unit, and algorithmic parallelism. Our approach for parallelized points-to analysis falls into the category of algorithmic parallelism. The simulated execution algorithm handles the three constructs that lead to independent nodes ((a) selection branches, (b) polymorphic calls, and (c) certain context definitions) in a way that facilitates parallel execution. For example, when the sequential simulated execution encounters a polymorphic call with multiple target methods it analyzes them in sequence. The parallelized simulated execution may process the target methods concurrently. When all target methods are analyzed both variants continue analyzing the SSA nodes succeeding the polymorphic call node. This recursive analysis style is the key property of the two algorithm variants. It forces the analysis of tasks (selection branch/target method/context) until they are stabilized before continuing with other parts of the program. It is possible for other data-flow analyses to use the same constructs for parallelism and the same approaches for creating independent tasks by plugging in other transfer functions, i.e., other computations for nodes. However, such a method must be a forward problem. Moreover, due to changing workloads in nodes, the thresholds may differ.

## 5. EVALUATION

All experiments were executed on a desktop PC with two Intel® Xeon® E5410 quad-core processors (i.e., eight cores) running at 2.33 GHz with 3.0 Gb of internal memory and using Windows XP SP3 and the Java™ version 1.6.0_17.

In the experiments we use a benchmark containing eight different Java programs ranging in size from 311 to 933 classes. All programs are listed in Table 1. The programs in the upper half of the table are taken from well-known test suites [2, 29, 8]. In the lower half, we have our own selection of “newer” test programs. We have picked programs larger than 300 classes, and freely available on the Internet. The programs that are included in the test suites but omitted here have very short analysis times, thus making them less interesting to consider. Table 1 also shows some basic information like the number of classes and reachable methods in these programs, and the time required for the points-to analysis using sequential simulated execution.

Finally, it shows a metric approximating the maximal expected speed-up in the respective benchmark program when eagerly parallelizing branches and polymorphic method calls. This metric is calculated from a statically created graph by emulating the parallelized simulated execution visiting each SSA node exactly once, i.e., optimistically assuming that each node takes one time unit to analyze and one update only. The metric is the average number of nodes that are processed at each time unit if there were an unlimited number of processors available. Under the above restriction, this is the maximal expected speed-up according to Amdahl's argument [1] and it is quite low, indicating that different parts of the program graphs are tightly coupled and very hard to parallelize.

### 5.1 Performance Metrics

The evaluation assesses the analysis time for each of the four heuristics of parallelized analysis and compares it to the analysis time for the sequential analysis for each benchmark program. The analysis times for the sequential analysis are reported in absolute numbers; cf. Table 1, while all other results are reported in relation to these absolute numbers. All data points are averages from three runs with the same settings, since times may fluctuate due to non-determinism.

The only parameter for Baseline is the number of threads \( p \) used in the thread pool. The parameters for the heuristics Threshold \( Th \) and Avoid Redundancy \( AR \) are \( p, T_b \) and \( T_p \); and the parameters for Reduce Redundancy \( RR \) are \( p, T_b, T_p \), and \( T_r \).

The three heuristics come in two variants. For the first variant (referred to as \( Th_0, AR_0, RR_0 \), resp., and named optimal) the parameters \( T_b, T_p, \) and \( T_r \) are optimized individually for each benchmark. For each benchmark, this variant uses the parameter which led to the shortest analysis time among the variety of parameters we tried out in the experiments, cf. the experimental setup below. The second variant (\( Th_g, AR_g, RR_g \), resp., named general) uses the same parameters of \( T_b, T_p, \), and \( T_r \) for all benchmarks. For \( Th_g \) (\( AR_g, RR_g \), resp.), the parameters were selected, which minimized the sum of the analysis times for all benchmarks in \( Th_0 \) (\( AR_0 \) and \( RR_0 \), resp.).

### 5.2 The Optimal Settings

Our first step was to run a series of experiments to find the optimal settings for the thresholds \( T_b, T_p, \) and \( T_r \), for each performance metric. The value range for the three

<table>
<thead>
<tr>
<th>Program</th>
<th>Classes</th>
<th>Methods</th>
<th>Time (s)</th>
<th>Maximal Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr2.7.7</td>
<td>396</td>
<td>2,685</td>
<td>22.7</td>
<td>2.43</td>
</tr>
<tr>
<td>javac1.4</td>
<td>538</td>
<td>3,906</td>
<td>65.8</td>
<td>2.36</td>
</tr>
<tr>
<td>javadoc1.3.1</td>
<td>452</td>
<td>2,347</td>
<td>11.1</td>
<td>1.33</td>
</tr>
<tr>
<td>soot-c</td>
<td>933</td>
<td>4,044</td>
<td>15.8</td>
<td>1.60</td>
</tr>
<tr>
<td>emma2.0</td>
<td>856</td>
<td>4,904</td>
<td>120.0</td>
<td>1.61</td>
</tr>
<tr>
<td>javacc3.2</td>
<td>311</td>
<td>2,136</td>
<td>2.1</td>
<td>1.21</td>
</tr>
<tr>
<td>jess4.5</td>
<td>364</td>
<td>1,825</td>
<td>3.1</td>
<td>1.52</td>
</tr>
<tr>
<td>pm33.2</td>
<td>556</td>
<td>3,320</td>
<td>2.6</td>
<td>1.60</td>
</tr>
</tbody>
</table>

Table 1: Benchmark programs and their statistics.
thresholds is \(0 \ldots NOM\), where \(NOM\) is the number of methods in each benchmark, i.e., between 1,825 and 4,904 (cf. Table 1). The value range for the number of threads \(p\) is 1..16 (twice the number of available cores).

An exhaustive search for finding optimal settings for the three parameters would require more than 16\( \cdot \)NOM\(^3\) (\(> 9.7 \cdot 10^{10}\) for \(jess\)) executions for each benchmark (\(RR_o\) and \(RR_p\)), which is not feasible. Therefore, our first decision to reduce the search space was to use a fix number of threads \((p = 12)\) for all setups. This number of threads was used in [9] and found to be suitable for a machine with 8 cores.

Furthermore, we started our optimization of the thresholds \(T_h\), \(T_p\) and \(T_o\) by running a first set of experiments using a sparse threshold grid \([0, 300, 600, \ldots, NOM]\). Based on these results, we made a second run using a more fine-grained grid \([0, 100, 200, \ldots, NOM]\) for \(T_h\), \(T_p\) and \(T_o\). We then chose values that showed the most promising for each benchmark. The actual results are based on the 343 \(T^3\) combinations closest to the settings for the champion in the first round (the settings used for the experiment with the lowest analysis time).

This means the combinations within three steps below and three steps above the champions’ settings for \(T_h\), \(T_p\) and \(T_o\).

The result, the parameter settings for the optimal variants, i.e., \(Th_o\), \(AR_o\) and \(RR_o\) are listed in Table 3.

The first thing to notice is that the best result is the fastest (i.e., highest speed-up) was always achieved by either \(Th_o\) or \(RR_o\). Thus, the \(AR\) approach where we strictly avoided any overlapping reachability sets in the independent tasks did not give the best results for any of the benchmark programs. Neither did Baseline.

Second, the average speed-up of about 1.76 using eight cores is not all that impressive. The light of the maximal expected speed-up of the problem, which is on average 1.71 over the benchmarks, it is quite good. Furthermore, the speed-up was in 4 (out of 8) cases larger than the maximal expected speed-up. This will be discussed in more detail in Section 5.4.

Finally, this first set of result was also used to compute the set of thresholds \(Th_o\), \(AR_o\) and \(RR_o\). These results are presented in the bottom of Table 3.

### 5.3 Number of Processed Methods

Table 2 shows the number of methods processed for Baseline, \(Th_o\), \(AR_o\) and \(RR_o\). The table lists a factor \(f = \frac{m_p}{\sigma_p}\), where \(m_p\) is the number of processed methods in the parallelized variant and \(m_o\) the number of processed methods in the sequential analysis. These numbers can be taken as a measure of the amount of work performed by each analysis. The results show that the work required to reach a fixed point can change from one approach to another.

The speed-up \(s_{x,p}\) of a parallel program \(\pi\) using \(p\) processors is defined as the performance \(t_o\) of the fastest known sequential program \(\sigma\) solving the same problem (executed on one processor) over the performance \(t_{x,p}\) of \(\pi\) executed on \(p\) processors.

According to Amdahl’s argument [1], the maximum speed-up achievable with any number of processors is bound by the inherent parallelism in the problem. As we will see in the experiments, cf. Section 5, our parallel approach sometimes performs less work (counted in number of analyzed methods) than the sequential approach. This causes the speed-up to be greater than a static measure of the inherent parallelism of the problem.

The reason for this is that we compare a sequential simulated execution \(\sigma\) with different variants of parallelized simulated execution \(\pi\). Observe that the order in which nodes are analyzed may differ between the sequential and the parallelized simulated execution. For instance, \(\sigma\) analyzes branches and polymorphic target methods in sequence, while the \(\pi\) may analyze them in parallel. Therefore, the values for nodes of different branches and target methods may interleave in \(\pi\).

Depending on the order in which the nodes are updated, the fixed point computations may take fewer or more iterations. Analysis values may only grow larger in the points-to analysis, i.e., get closer to the top element in the value lattice. Consequently, one and the same parallel program \(\pi\) may perform more or less work in different executions. Especially, executions of \(\pi\) may perform less work than its sequential counterpart \(\sigma\). Since the analysis values only may grow, it is more likely that they grow faster because of the parallel updating. A sufficiently small overhead for parallelization or a sufficiently large reduction of work explains a speed-up larger than the inherent parallelism.

### 5.4 Time Results

The detailed results from the seven variants are presented in upper half of Figure 2. The time values shown are relative to the analysis time for the sequential algorithm for each benchmark program. The lower half is a summary showing the average speed-ups for each individual approach.

At least one for each benchmark of the heuristics using optimal settings (\(Th_o\), \(AR_o\), \(RR_o\)) are effective for all eight benchmark programs, even though the time for \textsc{antlr2.7.7} is only reduced by 1%.

The results also show that three out of eight benchmark programs already benefit (\(< 100\%) from the simple Baseline parallel approach. However, Baseline comes
with a severe penalty (\(> 160\%\)) for certain benchmarks. The Threshold approach (\(Th_\alpha, Th_\beta\)) shows speed-up in most of the benchmarks and the fact that \(Th_\beta\) has a similar performance as \(Th_\alpha\) on average (1.76 vs. 1.51) indicates that it is rather insensitive to the chosen set of threshold parameters.

The RR approach is rather sensitive to the chosen threshold parameters. Its optimized variant \(RR_\alpha\) is together with \(Th_\beta\) the best performing heuristics in our experiments. Both have an average speed-up of 1.76. On the other hand, the general variant \(RR_o\) performs rather badly, indicating that RR requires quite a bit of fine-tuning before it can be used successfully.

Both version of AR show negative results on average (0.99 for \(AR_\alpha\) and 0.90 for \(AR_\beta\)). This indicates that strictly avoiding any overlapping reachability sets in the independent tasks is not a good idea.

In summary, the average speed-ups are not so impressive, but certain benchmarks (e.g., emma using \(Th_\alpha\)) shows a quite high speed-up (speed-up 4.43) indicating that parallelized algorithms for data-flow analysis can be of practical use in the future.

### 6. RELATED WORK

Most related efforts for parallelizing program analysis date back to the early 1990s, when parallel processing was dominated by distributed memory machines and, hence, only coarse-grained parallelism paid off.

Lee et al. [16] identify three different types of parallelism in data-flow algorithms: independent-problem, separate-unit, and algorithmic parallelism. They describe a unified framework that combines these three types. They use a hybrid algorithm [15] as an instance in the framework to exemplify its use. The basic idea is that the program graph is divided into strongly connected components (SCCs). Each SCC is analyzed in sequential processes communicating with other processes for exchanging analysis values between SCCs. For achieving better load balancing and coarser parallel programs, Lee et al. [17] suggested the parallel analysis of regions instead of single SCCs. Each region consists of one or several SCCs with a single entry node. Our analysis uses a more fine-grained parallelization approach where one SCC (of the call graph) is potentially analyzed by more than one thread.

Sgro and Ryder combine region and alias analysis [27] for achieving a balanced region size and reduced inter-region communication. Our approach uses dynamic load balancing instead and cares about reducing the interaction by avoiding/reducing synchronization.

Kramer et al. [14] exploit the inherent parallelism of independent paths in the control-flow graph when propagating data-flow information. Cyclic control-flow graphs are unrolled duplicating nodes in loops in such a way that backwards edges may be removed. A combination graph is created from the transformed control-flow graph, where combination nodes consist of sub-paths in the transformed control-flow graph. Our approach operates on standard program representations and does not unroll loops or duplicate nodes.

Edvinsson and L"owe [10] created a global program graph from the SSA graphs for the individual methods connected via call and return edges. They statically analyzed concrete program graphs wrt. the potential maximum inherent parallelism. They assumed parallelization along independent paths in the graphs, which are due to conditional statements (a) and polymorphic calls (b). Also, they statically estimated the scaling of a parallelization exploiting (a) and (b). This approach is experimentally evaluated [9] but only using parallelism from polymorphic calls. The experiments show that only five of eight benchmark programs benefit from the parallelization.

In contrast to our heuristics suggested in the present paper, only [9] of the previous approaches also exploits a combination of static and dynamic optimizations for achieving balanced load and reduced interaction between the parallel tasks. Likewise for our Reduce Redundancy heuristics, only [9] of the previous approaches also allows for redundant computation. However, in contrast to [9], the present paper shows that all benchmark programs benefit from a parallelized approach.

### 7. CONCLUSIONS AND FUTURE WORK

For adopting current developments towards architectures in commodity computer system containing more and more cores, we parallelized data-flow-base points-to analysis which is traditionally performed as a sequential fix-point algorithm. We exploited and evaluated vari-

<table>
<thead>
<tr>
<th>Program</th>
<th>(Th_\alpha) ((T_p, T_b))</th>
<th>(AR_\alpha) ((T_p, T_b))</th>
<th>(RR_\alpha) ((T_p, T_b, T_r))</th>
<th>Speed-up ((\text{Metric}))</th>
<th>Max. Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr2.7.7</td>
<td>(500, 1700)</td>
<td>(800, 900)</td>
<td>(800, 800, 500)</td>
<td>1.01 ((Th_\alpha))</td>
<td>2.43</td>
</tr>
<tr>
<td>javac1.4</td>
<td>(900, 700)</td>
<td>(800, 800)</td>
<td>(700, 700, 900)</td>
<td>2.07 ((Th_\alpha))</td>
<td>2.36</td>
</tr>
<tr>
<td>javadoc1.3.1</td>
<td>(1200, 800)</td>
<td>(600, 700)</td>
<td>(900, 700, 700)</td>
<td>1.42 ((RR_\alpha))</td>
<td>1.33</td>
</tr>
<tr>
<td>soot-c</td>
<td>(1100, 700)</td>
<td>(900, 700)</td>
<td>(700, 900, 900)</td>
<td>1.59 ((Th_\alpha))</td>
<td>1.60</td>
</tr>
<tr>
<td>emma2.0</td>
<td>(1200, 1000)</td>
<td>(700, 600)</td>
<td>(1000, 700, 900)</td>
<td>4.43 ((RR_\alpha))</td>
<td>1.61</td>
</tr>
<tr>
<td>javac3.2</td>
<td>(1000, 800)</td>
<td>(800, 900)</td>
<td>(700, 800, 700)</td>
<td>1.27 ((Th_\alpha))</td>
<td>1.21</td>
</tr>
<tr>
<td>jess4.5</td>
<td>(500, 1200)</td>
<td>(700, 900)</td>
<td>(900, 600, 600)</td>
<td>1.55 ((RR_\alpha))</td>
<td>1.52</td>
</tr>
<tr>
<td>pmd3.2</td>
<td>(500, 800)</td>
<td>(700, 800)</td>
<td>(700, 600, 600)</td>
<td>1.33 ((Th_\alpha))</td>
<td>1.60</td>
</tr>
</tbody>
</table>

Table 3: Settings for the three optimal variants \(Th_\alpha, AR_\alpha\) and \(RR_\alpha\).
ants parallelizing the points-to analysis of independent tasks due to (a) conditional statements, and (b) polymorphic method calls. The approaches are based on Simulated Execution used for fast and precise sequential static program analysis [18].

The parallelized points-to analysis’ performance was evaluated. The main result is that all analyzed benchmark programs save time with the parallelized approach compared to the sequential approach. We evaluate one baseline approach eagerly creating tasks at all given opportunities and three heuristics that control task generation based on work size and task redundancy. These four approaches come in two variants each: an optimal and a general. The optimal variant tunes parameters like number of threads, minimum workload, and maximum redundancy individually for each benchmark; the general variant uses the same (suboptimal) settings for all benchmarks. The evaluation was performed on eight Java benchmark programs. All eight benchmarks showed a speed-up using the optimal variant for the work-controlling heuristic Threshold, and seven out of eight benchmarks when using the general variant for Threshold. The benchmark (antlr2.7.7) that do not benefit from the general parallelization approaches only show an increase in execution time of less than 14%. The average speed-up over all benchmarks is 1.76 for the optimal variant using heuristics Threshold (1.51 for the general variant using heuristic Threshold). It is worth noting that points-to analysis is a whole program analysis. It is not possible to start multiple processes analyzing independent program parts in parallel. A whole program analysis considers the interaction between program parts and how they affect each other, directly and indirectly.

In the light of the analyzed inherent maximal expected speed-up in the benchmark programs of about 1.71 parallel activities on average at any time during the analysis using the simulated execution node update strategy, the speed-up is even higher than expected. This is explained by work reduction of the parallelized data-flow algorithms compared to the sequential counterpart.

Future work will aim at increasing the inherent maximal expected speed-up in the problems. The static estimates of the inherent maximal expected speed-up presented in our pre-study [10] scale better with the number of available cores, but are still limited to an average speed-up of 13 when using virtually arbitrarily many cores. Context-sensitive analysis is even more computationally intensive compared to the context-insensitive analysis as used in the present paper and would provide more opportunities for parallel processing. Context-sensitive analysis (c) is relevant since it enable more precise points-to analysis.

Using multi-threaded approaches in user-centric applications can infer problems if all cores are heavily used. Other applications may get reduced processor time and stop working as expected, such as loosing responsiveness. This can be mitigate by restricting the number of threads compared to the number of available cores. In a near future, there will be enough cores in processors so that several multi-threaded applications can co-exist without having problems with such starvation.

We will also improve finding the optimal or near-

Figure 2: Results of the parallelized approaches compared to the sequential analysis.
optimal settings for the parameters number of threads, minimum workloads, and maximum redundancy for a particular project. This was only sketched in this paper; we showed that general settings could be used and that they only infer a small negative effect on the efficiency compared to the optimal settings.

However, the efficiency of the parallelized approaches on a particular benchmark program depends on the optimum parameter setting, which, in turn, is (possibly) correlated to static properties of the program. The aim is to create a model based on these static program properties, such as cohesion and coupling between classes, recursive calls, and polymorphism. It can be used to calibrate the parameter settings without benchmarking. The foundation for this will be derived in future work from empirical studies correlating static program properties with optimal parameter settings in a larger set of benchmark applications.

8. REFERENCES
[27] V. Sgro and B. G. Ryder. Differences in Algorithmic Parallelism in Control Flow and Call


[29] SPEC JVM98 Benchmarks. 
http://www.spec.org/osg/jvm98.
