ParaSCAN: A Static Profiler to Help Parallelization

Ganesha Upadhyaya, Tyler Sondag, and Hridesh Rajan
TR14-06
Initial Submission: April 1, 2014
Revised: May 13, 2014

Abstract: This is the Technical Report version of the 2014 ICSME submission by the same title. It includes the ICSME version verbatim, followed by an appendix containing omitted contents.

Keywords: parallelism discovery and planning, software evolution

CR Categories:
D.3.3 [*Programming Languages*] Language Constructs and Features - Control structures

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

CONTENTS

ParaSCAN: A Static Profiler to Help Parallelization

Ganesha Upadhyaya
Iowa State University
ganeshau@iastate.edu

Tyler Sondag
Intel Labs
tyler.n.sondag@intel.com

Hridesh Rajan
Iowa State University
hridesh@iastate.edu

***Abstract*—Parallelizing software often starts by profiling to identify program paths that are worth parallelizing. Static profiling techniques, e.g. hot paths, can be used to identify parallelism opportunities for programs that lack representative inputs and in situations where dynamic techniques aren't applicable, e.g. parallelizing compilers and refactoring tools. Existing static techniques for identification of hot paths rely on path frequencies. Relying on path frequencies alone isn't sufficient for identifying parallelism opportunities. We propose a novel automated approach for static profiling that combines both path frequencies and computational weight of the paths. We apply our technique called ParaSCAN to parallelism recommendation, where it is highly effective. Our results demonstrate that ParaSCAN's recommendations cover all the parallelism manually identified by experts with 85% accuracy and in some cases also identifies parallelism missed by the experts.**

I. Introduction

Most legacy systems were not designed to be concurrent. The task of retrofitting concurrency to sequential programs is non-trivial. During the past three decades, many tools and techniques have been proposed to help programmers parallelize their software. State of the art tools and techniques can be broadly classified into three categories: automatic parallelizing compilers [1], [2], dynamic or profile-guided parallelization tools [3]–[5], and refactoring tools [6]–[8]. In spite of this abundance of tools and techniques, manual parallelization is still prevalent and often considered the best choice to achieve the desired benefits.

We believe that manual parallelization and parallelization tools in all three categories could benefit from offline information about parallelization candidates, e.g., methods, loops, or program paths, that are worth parallelizing. In other words, they can all benefit from static profiling for parallelism. In the absence of an efficient static profiler, automatic parallelizing compilers perform dependency analysis of the whole program; for larger programs this leads the problem of *path explosion* since the number of potential paths grows exponentially with the number of branches in the program [9]. Static profiling for parallelization could help dynamic and profile-guided parallelization tools with selecting representative inputs by giving them another metric for coverage (i.e., make sure that the inputs result in dynamic execution of the paths recommended by the static profiler). Finally, the reliance of refactoring tools on the programmer to select parallelization candidates can be decreased.

We propose a novel recommendation system called *ParaSCAN* to efficiently select candidates for parallelization using static profiling. ParaSCAN is built on our observation that parallelism opportunities that yield noticeable speedup often lie along average case execution paths (ACEPs). Unlike most likely paths [10] that focus purely on path frequency, ACEPs are paths that are likely both frequent and consume the lion's share of the program's execution time, as approximated by total execution cycles for the program statements in the path. Typically an ACEP is computed from execution traces gathered from many runs of the program for different representative inputs. Instead, ParaSCAN uses a new, completely static, technique for computing ACEPs, which is another contribution of this work.

To demonstrate the effectiveness of ParaSCAN's recommendations in helping the programmer for parallelization, we evaluated ParaSCAN on a total of 20 small, medium and large Java applications from Gang-Of-Four design pattern applications (GOFBench) [11], Java Grande [12], and NAS Parallel Benchmarks [13]. We compare our parallelization recommendations against manually parallelized code by experts. Our goal is to see how many of the manually parallelized methods can be automatically selected as parallelization candidates for inspection using static profiling.

Our results show that ParaSCAN's recommendations cover all the parallelism identified by experts. ParaSCAN's good accuracy (85%) helps compilers and refactoring tools to utilize ParaSCAN's recommendations without incurring much overhead. In terms of simplifying the analysis scope for compilers and tools, ParaSCAN on average reduced the program statements needed to be analyzed to approximately one third of the original program. ParaSCAN has 85% accuracy compared to the 37% of previous work in identifying parallelism opportunities. Further, parallelizing ParaSCAN recommendations for a real-world Java application, BiNA, produced 8-35% additional speedup over manually parallelized version.

II. Motivation

Imagine that a programmer is asked to parallelize the Biomolecular Network Alignment toolkit (BiNA) [14]. BiNA is used by molecular biologists to study the interaction patterns between different molecular participants such as genes, proteins and metabolites. It implements graph-kernel based algorithms for aligning large biomolecular networks by decomposing such networks into subnetworks and computing the alignment of the networks based on the alignment of the subnetworks.

The most common way to parallelize a program is incrementally through a set of behavior-preserving transformations,

i.e., a refactoring [6]. Refactoring is more economical than rewriting, however, it first requires identifying parallelism opportunities. This process is essentially a manual partial traversal of the program's callgraph. In a callgraph (CG), nodes are methods and directed edges represent calls. This task of browsing and understanding the details of the code often becomes fairly complicated. Figure 1 shows the CG for BiNA, which consists of 382 methods. For someone who didn't write the original version of the software, this is a daunting task, even for this medium-sized program.

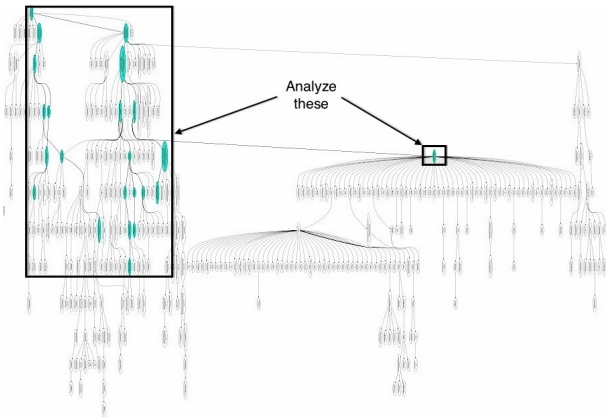


Fig. 1. Callgraph for BiNA showing ParaSCAN's recommendations consist of a much smaller subset of the callgraph (the highlighted nodes – only 6% of methods).

To reduce the burden on the programmer, ParaSCAN automatically browses the source code and presents the programmer with a significantly reduced portion of the program that it considers most beneficial to parallelize. In this example, rather than analyzing 382 methods the programmer can focus on only 23 methods (less than 6% of the total methods). Even if the consumer of the recommendations is not a person, this reduction in methods represents a significant reduction in work that must be done by subsequent analysis and/or optimization (e.g., parallelization).

III. ParaSCAN Overview

ParaSCAN's goal is to help programmers and tools by presenting a significantly reduced portion of the program that it considers most beneficial to parallelize. The key challenge in doing so is to statically determine the code portions that contain parallelization candidates. In this section, we briefly explain the key insights that are used by ParaSCAN to accomplish this task.

ParaSCAN takes a path-based static profiling approach for recommending parallelization candidates, because, paths (inter- and intra-procedural) encompasses both coarse- and fine-grained parallelism opportunities; at the same time, paths help ParaSCAN eliminate large portions of the code from consideration that lack beneficial parallelism opportunities. The challenge is to identify a subset of paths along which parallelization efforts will result in noticeable performance benefits. Our observation is that most beneficial parallelism opportunities lie along the *average-case execution paths* (ACEPs) of the program. ACEPs are the paths that have execution times within an acceptable range (Δ) of the average-case execution time (ACET). The ACET is the average time taken for a typical execution of the program. Typically an ACEP is computed from execution traces gathered from many runs of the program for different representative inputs. Instead, ParaSCAN computes ACEPs statically using a novel technique that combines both predicted path frequency and estimated path execution cycles. In doing so, the two challenges are: 1) *predicting path frequencies statically*, and 2) *determining the approximate execution cycles for each path statically*.

Predicting path frequencies statically is hard [15]. The state of the art technique for predicting path frequencies tends to ignore paths that have large impact on program state. For instance, paths that contain heavy computational statements. However, we believe that, paths that have large impact on program state should not be ignored in identifying parallelism opportunities because parallelizing these paths leads to large performance benefits. Hence, we propose a technique for predicting path frequencies using frequencies of statements in the path. The frequencies of statements predicted purely based on control-flow and type of statements ensures that we do not miss paths that have large impact on program state.

```
1  public synchronized V put(K key, V value) {
2   if (value == null) {// check for null value insertion
3    throw new NullPointerException();
4   }
5   if (count >= threshold) {// threshold is exceeded
6    rehash();
7   }
8   // create new entry and insert to the table
9   Entry<K,V> e = tab[index];
10  tab[index] = new Entry<K,V>(hash, key, value, e);
11  return null;
12 }
```

Fig. 2. The put method of Java SDK 1.6's `java.util.Hashtable` class. Some code has been omitted for illustrative simplicity.

To illustrate our technique, consider the example program shown in Figure 2 that has three paths. The path corresponding to the insertion of a new entry to the table (lines 9-11) is a frequent path. The other two paths corresponding to throwing an exception (lines 2-4) and rehashing (lines 5-7) are infrequent. This information is deduced by applying Ball and Larus heuristics [16]. The `if` block (lines 2-4) that compares a variable against `null` is less likely to be taken (pointer heuristics), hence, the exception path that contains it is also less likely. Similarly, the rehashing path that contains the `if` block (lines 5-7) that compares a variable against a constant is found to be less likely (opcode heuristics). In this way, using the frequency of program statements, we can predict the path frequencies.

ParaSCAN must also estimate computational weights of program paths. Using an instruction count that assigns equal weight to instructions is not accurate and execution time

captured by inserting timer based API calls is not possible in a static context. We propose a timing model that computes computational weight of statements in program paths. We use approximate execution cycles as computational weights that serves our purpose to differentiate different program paths.

Our intuition is that, for computing computational weights of larger instructions it suffices to know the computational weights of small subset of base instructions that formed larger instructions. As a foundation, we estimate the computational weight of individual machine instructions by approximating the relative complexity (in terms of execution cycles) of instructions with respect to one another. These estimates for individual instructions are then used to estimate computational weights for intermediate instructions, which are then used to estimate computational weights for entire program paths. The goal is to be accurate in terms of estimating which paths take significantly longer than others, hence, extreme precision of the timing model is not necessary. For instance, in our Java implementation of ParaSCAN, a path is decomposed into a set of bytecodes (complete decomposition is {path $\rightsquigarrow$ blocks $\rightsquigarrow$ statements $\rightsquigarrow$ bytecodes}) using intermediate representations of the program (Soot's Jimple/Baf representations [17]). We have built an interpreter based on Jikes RVM [18]'s runtime interpreter to translate Java bytecodes to machine codes. In a nutshell, by knowing the execution cycles for machine codes, path execution cycles are computed.

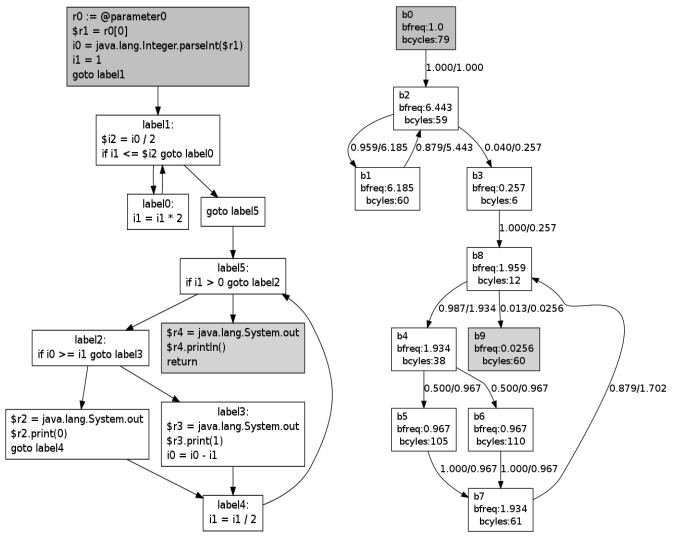
Upon computing the frequency and execution time (execution cycles $\times$ frequency) for program paths, we compute the ACET of the method by taking the weighted average of path execution times. More details of ACET computation are provided in Section V. For the example program shown in Figure 2, the path execution times for all three paths are compared against the ACET of the method to select ACEPs. A path is recommended as an ACEP only when its execution time is similar to ACET of the method. The most likely path of inserting a new hashtable entry (lines 9-11) is selected as the ACEP for the `put` method because it computationally is similar to the ACET of the method and also has high path frequency.

Likewise, we compute the ACET and select ACEPs for every method in the call graph by visiting methods in depth-first traversal (a fixed-point computation is used to handle recursion). The ACET of the callee method is aggregated in the caller method. We traverse the call graph again starting from the entry method and recommend the methods that lie along the ACEPs as parallelization candidates. ParaSCAN's output contains both methods and ACEPs in those methods (intra-procedural paths) as parallelization candidates. ParaSCAN's graphical output highlights selected methods in the call graph and ACEPs in the intra-procedural control-flow graphs.

IV. Background : Definitions

A **call graph** is a directed graph $CG = (V, E, v_0)$, where V is a set of nodes representing the procedures, $E \subseteq (V \times V)$ is the directed edge set of the graph where each edge (v_1, v_2) indicates that procedure v_1 calls procedure v_2, and $v_0 \in V$ denotes the entry procedure.

A **control flow graph** is a directed graph $CFG = (V, E, v_0, T)$, where V is a set of nodes representing the basic computational units such as statements or groups of statements, $E \subseteq (V \times V)$ is a directed edge set representing potential flows of execution, $v_0 \in V$ is the node representing the program entry point, and $T \subseteq V$, are the exit nodes.



(a) (b)

Fig. 3. (a) CFG of a Java program that prints the binary representation of an integer. Shaded blocks denote start and termination nodes. (b) The static profile information for the CFG shown in (a). The static profile has block frequency (bfreq) and block cyles (bcycles) for each block and the edges are labeled with the (edge probabilities/edge frequencies). For example, an edge label 0.959/6.185 indicates edge probability=0.959 and edge frequency=6.185

Given a control flow graph, $CFG = (V, E, v_0, T)$, a **path** is a sequence of vertices such that every two adjacent vertices are connected by an edge in E, v_0 is the start vertex, and the end vertex is any vertex in the set T. The set of such paths is defined as follows.

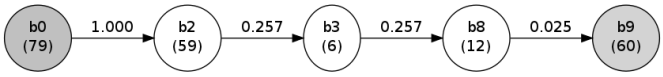
$$\mathcal{P}(\mathcal{CFG}) = \{\pi \mid \pi[0] = v_0 \wedge \pi[i] \in V \ \wedge \ \forall i \geq 0, (\pi[i], \pi[i+1]) \in E\} \quad (1)$$

V. ParaSCAN Static Profiler

ParaSCAN recommends parallelization candidates using its static profiler. The static profiler first computes the estimated ACET of the program, then identifies methods that are expected to contribute most to the ACET of the program, and finally selects average-case execution paths (ACEPs).

A. ACET Computation

We compute ACET for every method in the call graph by visiting in depth-first order. A method can be decomposed into a set of acyclic program paths using `CFG` of the method. A path can be described using path frequency indicating the frequency that the path will be taken and path execution cycles indicating the computational weight of the path. We now



(a) *Path-1*: Pathtime (PathFrequency * PathCycles) = 332,
PathCycles : (79 + 59 + 6 + 12 + 60) = 216,
PathFrequency : (1.000 + 0.257 + 0.257 + 0.025) = 1.539

Path-2: $\{b0 \rightarrow b2 \rightarrow b3 \rightarrow b8 \rightarrow b4 \rightarrow b5 \rightarrow b7 \rightarrow b8' \rightarrow b9\}$

Path-3: $\{b0 \rightarrow b2 \rightarrow b3 \rightarrow b8 \rightarrow b4 \rightarrow b6 \rightarrow b7 \rightarrow b8' \rightarrow b9\}$

Path-4: $\{b0 \rightarrow b2 \rightarrow b1 \rightarrow b2' \rightarrow b3 \rightarrow b8 \rightarrow b9\}$

Path-5: $\{b0 \rightarrow b2 \rightarrow b1 \rightarrow b2' \rightarrow b3 \rightarrow b8 \rightarrow b4 \rightarrow b5 \rightarrow b7 \rightarrow b8' \rightarrow b9\}$

Path-6: $\{b0 \rightarrow b2 \rightarrow b1 \rightarrow b2' \rightarrow b3 \rightarrow b8 \rightarrow b4 \rightarrow b6 \rightarrow b7 \rightarrow b8' \rightarrow b9\}$

Fig. 4. The six acyclic paths for the CFG shown in Figure 3(b). The shaded nodes are entry and exit nodes. Each path has nodes from b0 to b9. b0' indicates the second visit of b0 in case of loops. We show the computation of path cycles, path frequency and path time for *Path-1*. The values are calculated similarly for other paths.

describe the computation of path frequency and path execution cycle estimates.

For estimating path frequencies, we have extended Wu and Larus [19] static profile. Wu and Larus static profile is based on Ball and Larus [16] static branch prediction heuristics. The static profile consists of node and edge frequencies for `CFG` of every method where nodes are basic blocks containing a set of statements with a single entry point and a single exit point.

Roughly, we may think of the computation of block and edge frequencies as follows. Each branch direction has a "branch probability" as computed by the static predictor. Each edge has an edge frequency equal to the block frequency of the block at the source of the edge multiplied by the branch probability for this branch direction. Each block frequency is computed as the sum of incoming edge frequencies. Note that, the branch probabilities are computed directly by applying Ball and Larus heuristics. Also, Wu and Larus technique has proper treatment of loops while computing block and edge frequencies.

Consider Figure 3 (b) that shows the static profile for the CFG in Figure 3 (a). Here, the block frequency of b0 is 1 since b0 is the entry node. Consider edge $b2 \rightarrow b1$. The block frequency of source block $b2$ is 6.443 and the edge probability for this edge is 0.959 So, the edge frequency is $0.959 * 6.443 = 6.185$. Consider block $b2$, which has two incoming edges, $b0 \rightarrow b2$ with edge frequency 1.0, and $b1 \rightarrow b2$ with edge frequency 5.443. Hence the block frequency of $b2$ is $1.0 + 5.443 = 6.443$. Similarly block and edge frequencies are computed for rest of the program.

Using execution frequencies for blocks and edges, we obtain the path frequency estimates. The notion of path frequencies is defined in [15], [20]. In a nutshell, path frequency is the approximate number of times a path will be executed for various runs of the program. We compute ACET of a program as the weighted average of the execution time estimates of all acyclic program paths and path frequencies serve as the weight in this calculation.

Path frequency estimates. Every path, $\pi = \{\pi[0], \pi[1], \ldots, \pi[|\pi| - 1]\}$ has a set of vertices and set of edges. By knowing the frequency of edges (f_e), that formed the path, we can obtain the path frequency estimates as follows.

$$f_p(\pi, \mathcal{CFG}) = \Sigma_{i=0}^{|\pi|-1} f_e((\pi[i], \pi[i+1]), \mathcal{CFG}) \quad (2)$$

The idea behind summing the edge frequencies for computing path frequencies is as follows. A path consists of basic blocks and edges. At every branch, the taken and non-taken edges may have different frequencies. If the taken edge is part of the path, then the not-taken edge will be part of another path. So, by summing the edge frequencies at every basic block from the start block to the terminating block will give the path frequency.

As described in overview section, we use execution cycles as a measure of computational weight of program path. Our intuition is that, for estimating the execution cycles of larger instructions, it suffices to know execution cycles of small subset of base instructions. We have programmed our timing model with approximate execution cycles required for set of base instructions (machine codes). Given a path, we decompose it into set of base instructions. For instance, in Java implementation of ParaSCAN, every path goes through the decomposition ($\{\text{path} \rightsquigarrow \text{blocks} \rightsquigarrow \text{statements} \rightsquigarrow \text{bytecodes}\}$) and gets reduced to a set of machine codes. By using the execution cycles for machine codes, we compute execute cycles for every path.

Path execution cycles estimates. A path π can be decomposed into a set of blocks $\{\pi[0], \pi[1], \ldots, \pi[|\pi| - 1]\}$, where each block contains a set of statements. If $c_b(b)$ is the required execution cycles for any block b, then estimated path execution cycles (c_p) for any path π is defined as:

$$c_p(\pi) = \Sigma_{i=0}^{|\pi|-1} \; c_b(\pi[i]) \quad (3)$$

Thus far we have obtained the path frequency estimates and execution cycle estimates for paths. We now combine them to produce path execution time estimates.

Path execution time estimates. A path's execution time is computed as the product of the estimated path frequency and execution cycles. Given a path π, $\mathcal{CFG}$, $f_p(\pi, \mathcal{CFG})$ and $c_p(\pi)$, t_p is given by,

$$t_p(\pi) = f_p(\pi, \mathcal{CFG}) \times c_p(\pi) \quad (4)$$

Consider *Path-1* shown in Figure 4. Path execution cycles for this path is computed by summing the block execution cycles. That is, Path execution cycles of *Path-1* is $(79 + 59 + 6 + 12 + 60) = 216$. Path frequency of *Path-1* is computed by summing the edge frequencies. That is, path frequency of *Path-1* is $(1.0 + 0.257 + 0.257 + 0.025) = 1.539$. The path execution

time is $(1.539 \times 216) = 332$. Similarly, path execution time for all other paths is computed.

ACET. We compute the ACET of a method as the weighted average of the estimated execution times of all acyclic program paths. Estimated path frequencies serve as the weight in this calculation. Given a set of paths ($\mathcal{P}$) of a method with frequency ($f_p(\pi)$) and execution cycles ($c_p(\pi)$) for every path, the $ACET$ value for $\mathcal{P}$ is:

$$ACET(\mathcal{P}) = \frac{\Sigma_{\pi \in \mathcal{P}}\ t_p(\pi)}{\Sigma_{\pi \in \mathcal{P}}\ f_p(\pi, \mathcal{CFG})} \tag{5}$$

To illustrate, consider the CFG shown in Figure 3. The six acyclic paths for this CFG are shown in Figure 4, which also shows path execution time for *Path-1*. We calculated path execution times for other five paths and the sum of path execution time for these paths is 31658 cycles. The sum of the path execution frequency for these paths is 66.394. Thus, the ACET of this CFG of the method `main` is 476.82 cycles.

B. ACET Method selection

After computing ACET for all the methods in a call graph, our technique constructs a modified call graph that highlights "ACET methods" based on their ACET values. Starting with an entry point method, it selects the successor nodes in such a way that the most contributing successor is chosen. The idea being that the calling methods "contain" useful parallelism somewhere within them. Intuitively, the highlighted ACET methods are invoked in ACEPs of the parent method.

Given a call graph $CG = (V, E, v_0)$ and a real number $\Delta \in (0, 1]$, we define a modified call graph $CG' = (V', E', v_0)$ where,

- $V' \subseteq V$ is the set of nodes present in CG' such that, $\forall v \in V', \Delta \times ACET(v) \leq ACET(parent(v))$, and $\exists$ a path $\pi = (v_0, \ldots, v) \mid \forall i, \pi[i] \in V'$
- E' is the set of edges present in CG' where, $E' = \{(v_i, v_j) \mid (v_i, v_j) \in E$, and $v_i, v_j \in V'\}$,

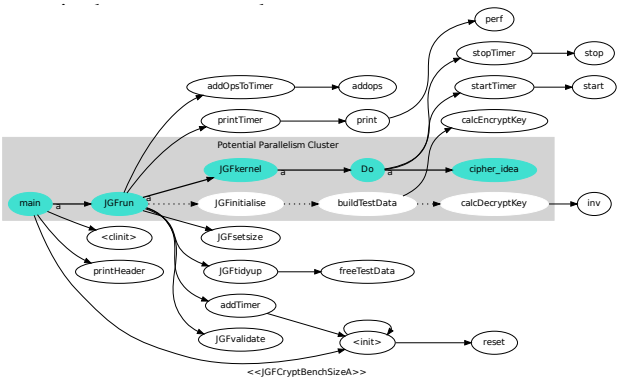


Fig. 5. The modified callgraph with "ACET methods" highlighted for the JG Crypt benchmark.

An example of such a modified call graph is shown in Figure 5. The highlighted region represents selected methods. The entry method `main` has ACET of 18780. In the next level, `printHeader` and `<clinit>` have low ACET, whereas, `JGFRun` has ACET of 17932. Hence `JGFRun` is selected and highlighted. Similarly, other methods are selected and highlighted. In a nutshell, the ACET method selection greatly reduces the burden of analyzing the methods which may not have enough computations to parallelize.

C. ACEP Computation

Every method highlighted previously has at least one intra-procedural program path. To identify average case execution paths (ACEPs) within a method, we select those paths whose execution time lies within +/-δ% of the ACET of the method. These are the paths which will likely be taken for representative inputs to the program and thus are of interest for detecting the parallelism opportunities.

Intra-procedural ACEP selection. We select paths that have their path execution time (c_p) similar to method's ACET. Given a path set, $\mathcal{P}$, $ACET(\mathcal{P})$, $c_p(\pi)$ for each path π and δ, we define ACEPs ($P \subseteq \mathcal{P}$) as follows.

$$P = \left\{ \begin{array}{l} \pi \mid \pi \in \mathcal{P} \\ \bigwedge \\ ACET'(\mathcal{P}) \leq c_p(\pi) \leq ACET''(\mathcal{P}) \end{array} \right\} \tag{6}$$

$$\text{where } ACET'(\mathcal{P}) = ACET(\mathcal{P}) - \delta, \text{ and}$$
$$ACET''(\mathcal{P}) = ACET(\mathcal{P}) + \delta$$

To illustrate, consider the paths shown in Figure 4. The ACET of the `main` method that contains these paths was 476.82. For, $\delta = 10\%$ of *ACET*, i.e., 47.68, the range for ACEPs is 429.14 to 524.5. Paths 2 and 3 have path execution times in this range and thus are selected as ACEPs for this method. Now, for $\delta = 20\%$ of *ACET*, i.e., 95.36, the range for ACEPs is 381.56 to 572.18. For this range, paths 2, 3, 5, and 6 are selected as ACEPs for this method. Note that, the value of δ is configurable and it has considerable impact on the accuracy of ParaSCAN's recommendations. In our experiments, we have used $\delta = 10\%$ which led us to not miss any potential parallelism opportunities with 85% accuracy.

D. Implementation Challenges

To evaluate our parallelism recommendation technique based on ACET, we have implemented ParaSCAN for Java [21]. The Java implementation of ParaSCAN has five components: pre-processing, loop analysis, path enumeration, ACET computation, and parallelism recommendation. Pre-processing constructs the call graph and control flow graphs of the input program statically using *Soot* [17] and its *Spark points-to-analysis* [22]. The loop analysis stage statically estimates loop bounds using the technique described by Michiel *et al.* [23]. The path enumeration stage computes all acyclic intra-procedural program paths [15]. The ACET computation stage computes the ACET and ACEPs for every method using the static profile and approximate execution cycles (provided by our timing model). Finally, the parallelism recommendation stage recommends methods and ACEPs of the methods as parallelism candidates.

Challenges. The selection of an efficient points-to analysis framework handles the dynamic aspects of object-oriented languages. For instance, in our Java implementation of ParaSCAN, we use the points-to analysis provided by the Spark framework [22] to compute the types of objects that may be referenced by each variable, then based on the type information of the receiver variable, a possible target for each virtual call can be computed more precisely. This way, Soot's Spark framework provides accurate enough call graphs.

To deal with Java reflection, ParaSCAN requires that a reflection trace file that contains information about reflective calls is provided. ParaSCAN supplies this trace file to Soot while building call graphs and control flow graphs to resolve reflective call sites.

Even though we enumerate acyclic program paths, program paths that go through the loop body are treated specially. For determining statically available loop bounds, we perform a non-trivial flow analysis using points-to-analysis information provided by Soot's Spark framework. Like most static analyses, loop bounds that are input-dependent remain undeterminable, however, we include paths that go through the loop with unknown bound as ACEPs in ParaSCAN for mitigating the risk of missing parallelism opportunities.

VI. Evaluation

In this section, we first evaluate the accuracy, scalability and benefits of ParaSCAN. We then, compare ParaSCAN against hot paths technique for identifying parallelism opportunities. Finally, we present a case-study to demonstrate ParaSCAN's applicability on real-world programs. In the rest of this section, *manual parallelism* refers to parallelism identified by the experts and *parallelism locations* refers to methods. ParaSCAN's recommendations are both methods and ACEPs (intra-procedural paths) of the selected methods. However, we evaluate only recommended methods because accurately comparing ACEPs against intra-procedural paths that contains parallelism in manually parallelized code is difficult.

A. Evaluation Methodology

We have selected a set of benchmarks that all meet two criteria. First, both serial and parallel versions of the benchmark are available. This allows us to avoid bias in evaluating ParaSCAN's recommendations, since the parallel versions were not developed specifically for these experiments, or by the authors. Second, the parallel version is not just multiple iterations of the serial code, because in that case recommendation is trivial. The DaCapo 9.12 [24] and SpecJVM 2008 [25] benchmarks are omitted because of the second criteria.

We use Gang-Of-Four design pattern applications (GOFBench) [11], Java Grande [12], and NAS Parallel Benchmarks [13] to evaluate our parallelism recommendations. GOFBench is a collection of 18 small to medium scale applications that use an implicitly concurrent framework for parallelization. Out of these, we use applications for chain of responsibility, composite, decorator, facade, and visitor design patterns as representative benchmarks for evaluating our recommendation system (others were omitted since recommendations were simple and/or obvious). We use sections two and three for the Java Grande benchmarks (because both parallel and serial versions are available), and NAS parallel benchmarks version 3.0 (the only Java version available).

B. Evaluation of Recommendation Precision

Our main claim is that ParaSCAN's recommendations are precise. This is important because too many extra recommendations can substantially increase the stakeholder's total cost of parallelization efforts.

RQ1: Does ParaSCAN help precisely identify potential parallelism?

To answer this question, we ran ParaSCAN on the serial version of a benchmark and then compared its recommendation with the manually parallelized version of the benchmark. The comparison will result in one of three cases. First, the recommended parallelism location is also a manually parallelized location. Second, the recommendation system misses a manually parallelized location. Third, the recommendation system recommends an additional parallelism location. The following three metrics capture these three outcomes.

Recommendation Precision Metrics.

- *Precision* is defined as the fraction of the recommended methods that match the manually parallelized methods.
- *Recall* is the fraction of the manually parallelized methods that match the recommended methods.
- *F-score* is the harmonic mean of precision and recall.

Precision ensures that the system does not incur too much overhead by recommending unnecessary methods. Recall ensures that not many parallelism opportunities are missed. F-score reflects the accuracy of the recommendation system. These three metrics are computed using standard definitions as shown in Figure 6.

Results. The results are shown in Figure 6. The column marked *#mt* shows the number of methods recommended by ParaSCAN for each of these 20 programs. The column marked *#me* shows the *best case estimation* of the number of methods recommended by an expert. This is computed by counting the number of methods in the call graph from the entry point of the program to the method that contains the parallelization point in the parallel version of the same benchmark.

Analysis. ParaSCAN achieves 100% recall by slightly reducing the precision (77%) to ensure that it does not miss any manually parallelized methods. We analyze the extra recommended methods that account for slightly low precision and demostrate that they contain additional parallelism opportunities and they are not just overheads. F-score of 100% indicates perfect recommendations, which our system achieves in 7 cases. Figure 7 shows one such case. On average we obtain an F-score of 85%. For 13 of 20 benchmarks, ParaSCAN recommended extra methods.

Figure 8 shows one such recommendation. For this benchmark, ParaSCAN recommended additional methods as parallelization candidates compared to the manually parallelized

Legend		
	m = Total No. of methods	l = No. of Loops
	mt = No. of methods in recommendation	me = No. of methods in expert recommendation
	m- = No. of methods missed	m+ = No. of extra methods in recommendation
	$t0$ = the preprocessing time	$t1$ = the analysis time
	t = the total time taken by ParaSCAN	

	Parallelizable	*Not Parallelizable*
Recommended	true positive (me)	false positive (m+)
Not Recommended	false negative (m-)	true negative ($m - me$)

Precision (P) = me / (me + m+)

Recall (R) = me / (me + m-)

F-score (F) = 2 × (Precision * Recall) / (Precision + Recall)

				ParaSCAN							HotPaths							ParaSCAN(time)		
	Name	m	l	mt	me	m-	m+	P	R	F	mt	me	m-	m+	P	R	F	t0	t1	t
GOF	Composite	11	1	2	2	0	0	100%	100%	100%	15	2	0	13	13%	100%	24%	73s	2s	75s
	Visitor	16	0	5	5	0	0	100%	100%	100%	14	5	0	9	36%	100%	53%	74s	6s	80s
	CoR	55	1	4	4	0	0	100%	100%	100%	1	0	4	1	0%	0%	0%	76s	15s	91s
	Decorator	15	0	7	6	0	1	86%	100%	92%	14	6	0	8	43%	100%	60%	74s	6s	80s
	Facade	11	0	4	4	0	0	100%	100%	100%	14	4	0	10	29%	100%	44%	74s	5s	79s
Java Grande	Crypt	30	17	8	5	0	3(3)	63%	100%	77%	2	0	5	2	0%	0%	0%	72s	9s	81s
	LUFact	33	36	7	4	0	3(2)	57%	100%	73%	8	1	3	7	12%	25%	17%	72s	10s	82s
	Series	28	6	4	4	0	0	100%	100%	100%	1	0	4	1	0%	0%	0%	72s	10s	82s
	SOR	26	7	4	4	0	0	100%	100%	100%	5	4	0	1	80%	100%	89%	74s	9s	83s
	Sparse	27	5	5	5	0	0	100%	100%	100%	1	0	5	1	0%	0%	0%	74s	10s	84s
	Moldyn	35	28	9	6	0	3(3)	67%	100%	80%	4	3	3	1	75%	50%	60%	75s	15s	90s
	MonteCarlo	110	17	12	5	0	7(6)	42%	100%	59%	5	0	5	5	0%	0%	0%	75s	18s	93s
	RayTracer	68	10	6	4	0	2(2)	67%	100%	80%	7	4	0	3	57%	100%	73%	73s	11s	84s
NAS Bench	BT	44	396	7	6	0	1(1)	86%	100%	92%	20	6	0	14	30%	100%	46%	93s	229s	322s
	CG	31	54	4	2	0	2(2)	50%	100%	67%	10	2	0	8	20%	100%	33%	74s	33s	107s
	FT	37	163	4	3	0	1(1)	75%	100%	86%	12	3	0	9	25%	100%	40%	75s	45s	120s
	IS	27	17	4	3	0	1	75%	100%	86%	12	3	0	9	25%	100%	40%	76s	27s	103s
	LU	44	229	8	3	0	5	38%	100%	55%	22	2	1	21	9%	67%	15%	97s	309s	406s
	MG	41	337	8	4	0	4	50%	100%	67%	21	3	1	18	14%	75%	24%	76s	40s	116s
	SP	44	445	7	6	0	1(1)	86%	100%	92%	16	6	0	10	38%	100%	55%	94s	313s	407s
							Average	77%	100%	85%					25%	66%	37%	77s	57s	134s

Fig. 6. Analysis results for GOFBench, Java Grande, and NAS benchmarks [Note: m+ column of ParaSCAN contains numbers like 3(2) indicates, out of 3 extra methods, 2 were recognized to have fine-grained paralleism opportunities]

code. To further investigate whether some benefit could be obtained by parallelizing these suggestions, we manually parallelized these methods. The result of this parallelization showed that paying heed to ParaSCAN's recommendations has value. We saw an additional speedup of about 12.5% for this already vetted benchmark.

Another interesting result that we saw in Java Grande and NAS benchmarks was that, *more fine-grained parallelism was available but not exploited*. Note that both benchmark suites use Java threads as the main parallelization mechanism. GOF benchmark uses the Java Fork/Join Framework [26]. Since threads are known to incur substantial overhead in creation and startup, experts omitted parallelized locations in Java Grande and NAS that they thought would not be worth that overhead. *ParaSCAN identified these locations.* Figure 9 shows an example.

Figure 9 shows ParaSCAN's recommendations for RayTracer benchmark, where it recommended methods `trace` and `shade` also to be parallelized. We examined the code and found them to indeed be parallelizable, however, a more fine-grained parallelization mechanism that has a lower task creation and startup overhead, e.g., task-based parallelism in ForkJoin framework [26], is needed to see substantial speedup. In Figure 6, *#m+* column under ParaSCAN contains these results enclosed inside parenthesis. We found that for 7 of 13 benchmarks, where ParaSCAN reported extra methods, all extra recommendations were candidates for fine-grained parallelization based on Lea's criteria [26]. We did not explore this further as improving parallelism in these benchmarks was not our central goal.

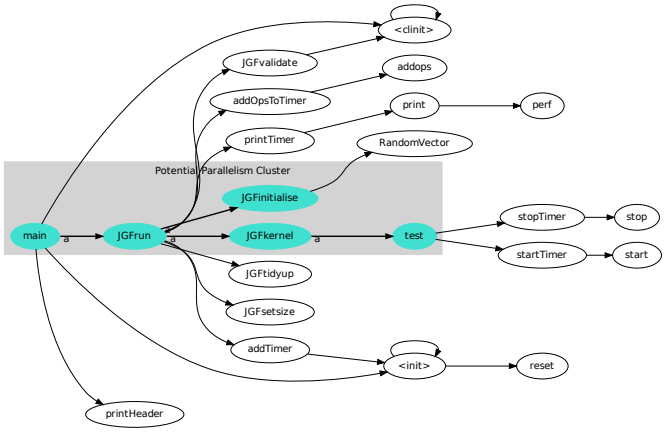


Fig. 7. The highlighted call graph for Java Grande Sparse Matrix multiplication. Colored nodes represent the expert recommendation. The grey box is ParaSCAN's recommendation.

For remaining 5 benchmarks: Decorator, MonteCarlo, IS, LU, and MG, ParaSCAN recommended extra locations. Each

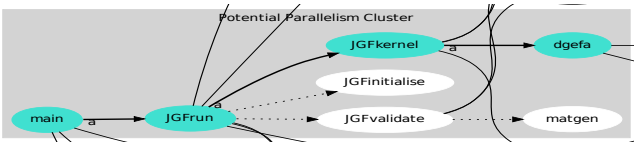


Fig. 8. The highlighted and elided call graph for JG LUFact benchmark. The colored nodes represent the expert recommendation.

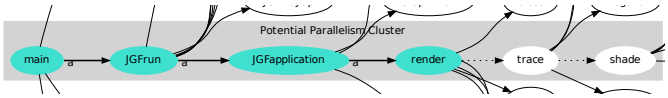


Fig. 9. The highlighted and elided callgraph for JG RayTracer.

case is now briefly described. For the Decorator benchmark, since ACETs were smaller, ParaSCAN was unable to filter out this false positive. For the MonteCarlo benchmark, an extra recommendation was the method `processResults` that contained a computationally heavy for loop with bounds given by a field in the containing class `AppDemo`. This field is set to a small constant by another class `CallAppDemo`, which is a client of the class `AppDemo`. So, human experts may have been able to deduce that this loop will only run a small number of times, whereas ParaSCAN did not deduce this and thus provided `processResults` as one extra recommendation. A similar method, `initKeys`, was presented as a recommendation for the IS benchmark.

For the LU and MG benchmarks, ParaSCAN recommended 5 and 4 additional methods respectively. We examined these methods. All of these methods are computationally heavy (e.g., one of these methods `erhs` in LU contains 41 loops). However, they are also dependency heavy (e.g., code in most of these 41 loops in `erhs` reads/writes one or more of 44 variables). Thus, parallelization of these methods poses serious risks of introducing inadvertent data races in the program. We believe, but do not have evidence to confirm or deny, that this may be the reason to run these methods serially in both LU and MG benchmarks.

In summary, for a set of 20 studied benchmarks, ParaSCAN achieved 77% precision, 100% recall, 85% F-score. Also, most of the extra recommendations actually turned out to be genuine parallelism that may have been missed or intentionally omitted by human experts.

C. Evaluation of Benefits

Our second claim is that ParaSCAN's recommendations are beneficial to the stakeholders (programmer or parallelizing compilers) by reducing the scope of dependency analysis.

RQ2: Does ParaSCAN help decrease the scope of dependency analysis for parallelizing compilers?

To answer this question, we measured the number of program statements in ParaSCAN recommendations and compared it to total statements for each benchmark program. Although in practice programmers may not analyze the entire program, this comparison reflects a reduction in scope of the initial set of locations they will begin with.

Scope Reduction Metrics.

- *%Scope Reduction (#Statements)*: is defined as the percentage reduction in program statements which need to be analyzed after using ParaSCAN recommendations.

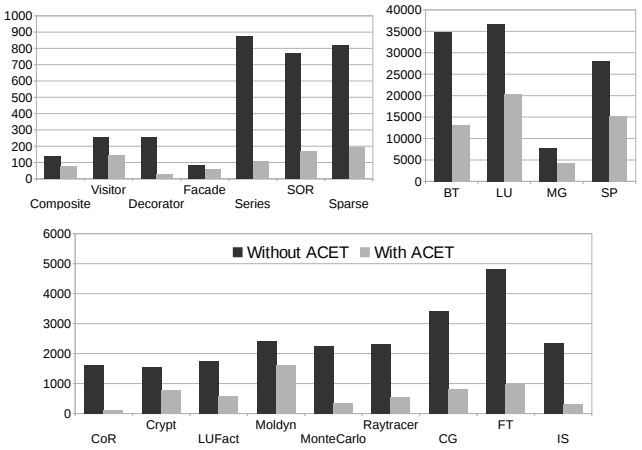


Fig. 10. Scope reduction w.r.t. program statements (y-axis) for GOF-Bench, Java Grande and NAS Parallel benchmarks; on average, 65% reduction in statements.

Results and Analysis. Figure 10 shows the parallelism analysis scope reduction with respect to program statements for Java Grande and NAS Parallel benchmark programs. The ACET analysis and ACEP selection used in ParaSCAN greatly reduced the scope of parallelism detection. The average 65% reduction in program statements helps to decrease the efforts towards parallelism detection and directly impacts the recommendation time, which is evaluated in next section.

D. Evaluation of Scalability

Our final claim is that ParaSCAN produces recommendations in a time proportional to the complexity of software.

RQ3: Does ParaSCAN produce recommendations in a timely manner?

Metrics. We answer this question by measuring the time taken for the complete analysis. We measure three time components: *t0* is defined as the time taken by pre-processing stage, mainly to generate call graph and control flow graphs, *t1* is defined as the time for ACET analysis and parallelism detection together, and *t* is defined as the overall recommendation time. Figure 6 shows these metrics and their means.

Analysis. The pre-processing time (*t0*) is relatively stable. Since this time is heavily dependent upon Soot's implementation, we expect this to also improve as Soot improves. For GOF and Java Grande benchmarks, which have few loops, preprocessing time dominates the total analysis time (*t*) . For NAS benchmarks, which have higher numbers of loops, the ACET analysis and parallelism recommendation time (*t1*) dominates. Further analysis shows that majority of this time is spent in loop analysis and bound computation. We plan on improving these analyses to further improve ParaSCAN's scalability.

E. ParaSCAN Vs Hot Paths

Our claim is that hot methods and hot paths generated using the state of the art static hot paths technique is not sufficient to identify parallelism locations.

RQ4: Can hot paths technique be used instead of ParaSCAN?

Methodology. For the benchmark programs listed in Figure 6, we collected hot methods using an implementation [27] of Buse and Weimer [15] hot paths identification algorithm. This implementation reports over 90% accuracy in hot path detection. We then compared hot methods against methods that are parallelized in the manually parallelized versions of the benchmarks. The *HotPaths* column in Figure 6 lists the results of this comparison. We compute *precision*, *recall*, and *F-score* for the hot paths technique similar to ParaSCAN. We then compare these metrics with those computed for ParaSCAN.

Analysis. When compared to ParaSCAN (precision:77%, recall:100%, F-score:85%), the hot paths technique (precision:26%, recall:66%, F-score:37%) is less suitable for static parallelism detection. We now discuss these results in detail.

For 11 out of 20 benchmarks the hot paths technique did not miss any methods that are recommended by experts for parallelization (recall of 100%). For 4 benchmarks the hot paths technique missed all methods that are recommended by experts for parallelization (recall 0%). For the remaining 6 benchmarks, the hot paths technique missed some methods. On average hot paths achieves 66% recall for the benchmarks we have evaluated. After investigating further, we found that these missed methods can be captured as hot methods at lower accuracy settings. Meaning, by allowing more unnecessary methods (methods that are not recommended by experts), it can be ensured that parallelism opportunities are not missed, however, at the cost of accuracy and precision. In summary, the low precision and accuracy of the hot paths technique suggests that it may not be a good choice for identifying parallelism opportunities statically over ParaSCAN. Note that, we did not compare the recommendation time of the hot paths technique against ParaSCAN because ParaSCAN produces recommendations fairly quickly and the hot paths technique on average takes about the same or slightly less time as ParaSCAN.

F. Threats to Validity

The use of manually parallelized versions of benchmarks to test ParaSCAN's recommendations poses two threats. First, program paths in the serial and the parallel version may be substantially different. Second, human experts may have missed some parallelization opportunities. To tackle the first threat, we compare paths in two versions to check whether the prefix of paths leading up to the parallelization point are identical. To tackle the second threat, we manually analyze parallelism recommendations where it differs.

Another threat to validity is our selection of benchmarks. The Gang-Of-Four (GOF) design pattern framework serial versions might have been implemented with the concurrent version in mind. To reduce the risk of this threat, we further evaluate using the well known parallel benchmark suites Java Grande and NAS Parallel Benchmarks (NPB) that also have both serial and parallel versions.

G. Case Study

In this section we demonstrate ParaSCAN's application to a non-trivial Java program, the BiNA framework, [14] and demonstrate that parallelizing the recommendations lead to considerable speedup. The Biomolecular Network Alignment (BiNA) Toolkit is a framework for studying biological systems at the molecular level such as genes, proteins and metabolites. Molecular biologists use BiNA for studying the interaction patterns between various molecular participants. BiNA compares and aligns interaction patterns among a large number of molecular participants. BiNA implements two graph-kernel based algorithms to decompose the network, cluster-based and k-hop neighborhood. It uses a divide-and-conquer approach to align the large biomolecular networks by decomposing them into sub-networks and computing the alignment of the networks based on the alignment of the sub-networks.

The original implementation of BiNA used explicitly created threads for computing the alignment of the sub-networks. We removed explicit threading from BiNA's original implementation to create a sequential version of BiNA. We ran ParaSCAN on the sequential version of BiNA to get parallelism recommendations. Figure 1 shows ParaSCAN recommendations for BiNA. The call graph shown in the figure contains 382 methods of which 23 methods are recommended by ParaSCAN. These recommendations mainly fall along two paths in the call graph. These two paths correspond to the two different algorithms to decompose the network, cluster-based and k-hop neighborhood. In both of these algorithms, the network is decomposed into subnetworks and the alignment score is computed to study the interaction patterns.

We compared ParaSCAN recommendations against the explicitly parallelized methods of the original BiNA implementation. We found that, all explicitly parallelized methods in the original BiNA implementation are covered by ParaSCAN recommendations. We explored the extra recommendations made by ParaSCAN to determine if the extra recommendations are valid parallelism opportunities or just false positives. One such recommendation corresponds to computing the shortest path distance between every pair of nodes in the network. The other recommendations are in the creation of the adjacency graphs where the input file containing a large number of strings is parsed to construct the network. These extra recommendations correspond to the parallelization opportunities in the enhanced version of BiNA reported by Rajan *et al.* [11]. Rajan *et al.* showed that by parallelizing these extra recommendations a speedup of 8-35% is achieved over original threaded version of BiNA. To summarize, ParaSCAN's recommendations covered all methods manually parallelized by experts as well as additional opportunities that, when parallelized, give additional noticeable speedup.

VII. Related Work

Automatic parallelizing compilers [1], [2]. Given a sequential program these compilers statically analyze the program, determine parallelization candidates, and create the parallel program automatically. Candidates are selected by statically analyzing the control and data dependencies between code segments. A major source of difficulties with automatic parallelization is when the program is large and has many branches; they suffer from the problem of *path explosion* since analysis of the dependencies of all program paths is required. To remedy this problem, automatic parallelizing compilers could use ParaSCAN to initially filter code segments that are worth parallelizing and perform dependency analysis only on these segments.

Refactoring tools [6]–[8]. These tools provide a way to actively involve the programmer in the parallelization process. They rely on the programmer to select the parallelization candidates upon which the tool statically check if it is safe to parallelize (using dependency analysis) and generates parallel program. ParaSCAN can help these tools and developers using these tools select parallelization candidates.

Dynamic parallelization tools [3]–[5]. These tools run the program on representative inputs, analyze data and control dependencies in profiled program paths to identify opportunities for parallelism. Dynamic approaches can identify more parallelism opportunities, however they require that the program is run on all representative inputs. Many studies have reported that program inputs impose a strong influence on program behavior [28], [29]. Shen and Mao [29] have shown that, the behavior defining the execution frequency of statements can vary largely for different inputs. Often developers using these tools are required to verify the parallelization candidates. When compared to these tools, ParaSCAN identifies methods and program paths that are worth parallelizing statically, without requiring representative inputs. We believe ParaSCAN's recommendations could be used by these tools and developers to help develop representative inputs by ensuring that the inputs result in dynamic execution of the paths recommended by ParaSCAN.

Profiling tools. Functions that dominate the execution time of the program, hot functions, are the candidates for parallelization. A profiler that captures the program execution traces could be used, however, the limitations of representative input and input sensitivity are inevitable. A static profiling technique, hot paths [15], generates most frequent program paths and methods based on predicted frequency. Their work assigns low frequencies to paths that have large impact on program state; for instance, a path that contains heavy computational statements. In contrast, ACEPs are paths that are frequent and includes heavy computations so that parallelizing them will result in performance benefits. When compared to hot paths technique, ParaSCAN selects parallelization candidates with better accuracy as shown in our experiments.

Acknowledgment

This work was supported in part by the US NSF under grants CCF-11-17937 and CCF-08-46059. We thank Robert Dyer for comments.

References

[1] K. Beyls, E. D'Hollander, and Y. Yu, "JPT: a Java parallelization tool," *Recent Advances in PVM/MPI, Springer'99*.
[2] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira, "Automatic loop transformations and parallelization for Java," in *Proceedings of ICS'00*.
[3] Z. Li, A. Jannesari, and F. Wolf, "Discovery of potential parallelism in sequential programs," in *Proceedings of ICPP'13*.
[4] C. Hammacher, K. Streit, S. Hack, and A. Zeller, "Profiling java programs for parallelism," in *Proceedings of ICSE'09 MSE Workshop*.
[5] K. Streit, C. Hammacher, A. Zeller, and S. Hack, "Sambamba: A runtime system for online adaptive parallelization," in *Proceedings of Compiler Construction, Springer'12*.
[6] D. Dig, "A refactoring approach to parallelism," *IEEE Software Journal, 2011*.
[7] J. Wloka, M. Sridharan, and F. Tip, "Refactoring for reentrancy," in *Proceedings of the 7th joint meeting of the ESEC/FSE'09*.
[8] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson, "Relooper: refactoring for loop parallelism in Java," in *Proceedings of OOPSLA'09*.
[9] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems, Springer'08*.
[10] T. Ball and J. R. Larus, "Using paths to measure, explain, and enhance program behavior," *IEEE Computer Journal'00*.
[11] H. Rajan, S. M. Kautz, and W. Rowcliffe, "Concurrency by modularity: Design patterns, a case in point," in *OOPSLA/Onward!*, 2010.
[12] L. Smith, J. Bull, and J. Obdrizalek, "A parallel Java Grande benchmark suite," in *ACM/IEEE Conf. on Supercomputing*, 2001, pp. 6–6.
[13] M. Frumkin, M. Schultz, H. Jin, and J. Yan, "Implementation of the NAS Parallel Benchmarks in Java," 2002.
[14] F. Towfic, M. H. W. Greenlee, and V. Honavar, "Aligning biomolecular networks using modular graph kernels," in *Algorithms in Bioinformatics, Springer'09*.
[15] R. Buse and W. Weimer, "The road not taken: Estimating path execution frequency statically," in *Proceedings of ICSE'09*, pp. 144–154.
[16] T. Ball and J. R. Larus, "Branch prediction for free," in *Proceedings of PLDI'93*, pp. 300–313.
[17] R. Vallee-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot-a Java bytecode optimization framework," in *the conference of the Centre for Advanced Studies on Collaborative research*, 1999, p. 13.
[18] IBM, "Jikes Research Virtual Machine (RVM)," http://jikesrvm.sourceforge.net/.
[19] Y. Wu and J. Larus, "Static branch frequency and program profile analysis," in *Proceedings of symp. on Microarchitecture, ACM'94*.
[20] T. Ball, P. Mataga, and M. Sagiv, "Edge profiling versus path profiling: The showdown," in *Proceedings of symp. on PPL'98*.
[21] ParaSCAN, "A Static Profiler for Parallelization," http://design.cs.iastate.edu/parascan/.
[22] O. Lhotak, "Spark: A flexible points-to analysis framework for Java," *Master's thesis, School of Computer Science, McGill University, Montreal, Canada*, 2002.
[23] M. d. Michiel, A. Bonenfant, H. Cass'e, and P. Sainrat, "Static loop bound analysis of C programs based on flow analysis and abstract interpretation," in *Proceedings of RTCSA'08*, pp. 161–166.
[24] S. M. Blackburn *et al.*, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proceedings of OOPSLA'06*.
[25] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko, "SPECJVM2008 performance characterization," *Computer Performance Evaluation and Benchmarking*, pp. 17–35, 2009.
[26] D. Lea, "A Java Fork/Join Framework," in *Java Grande*, 2000.
[27] K. Ali, "Using bayesian learning to estimate how hot an execution path is," *cs886, Fall 2010 Report, University of Waterloo*.
[28] K. Tian, Y. Jiang, E. Z. Zhang, and X. Shen, "An input-centric paradigm for program dynamic optimizations," in *ACM Sigplan Notices, 2010*.
[29] X. Shen and F. Mao, "Modeling relations between inputs and dynamic behavior for general programs," in *Proceedings of LCPC, Springer'08*.