Profiling Java Programs for Parallelism

Clemens Hammacher, Kevin Streit, Sebastian Hack, Andreas Zeller
Department of Computer Science
Saarland University
Saarbrücken, Germany
{hammacher, streit, hack, zeller}@st.cs.uni-saarland.de

Abstract

One of the biggest challenges imposed by multi-core architectures is how to exploit their potential for legacy systems *not built with multiple cores in mind. By analyzing* dynamic data dependences *of a program run, one can identify independent computation paths that could have been handled by individual cores. Our prototype computes dynamic dependences for Java programs and recommends locations to the programmer with the highest potential for parallelization. Such measurements can also provide starting points for automatic, speculative parallelization.*

1. Introduction

A central challenge of multi-core architectures is how to leverage their computing power for systems that were not built with parallelism in mind—that is, the vast majority of programs as we know them. Recent years have seen considerable efforts in *automatic parallelization,* mostly relying on *static program analysis* [3] to identify sections amenable for parallel execution, and using concepts like *transactional memory* to preserve data dependences [10].

While these efforts have shown impressive advances, we believe that they will face important scalability issues. The larger a program becomes, the harder it gets to precisely identify dependences between items—not to speak of multi-language code, or code that already has basic parallelism. At the same time, however, it may well be that it is *large systems* that offer the highest potential for parallelization. This is due to essential principles of software design: High abstraction, encapsulation, modularity, and hierarchy all keep computation local, while minimizing global interaction between components—which implies that such local computations could also be executed in parallel.

To further explore this hypothesis, we wanted to *measure* the potential for parallel execution in recent object-oriented programs. Our approach works as follows:

```
1   void main(String[] args) {
2       int n = parseInt(args[0]);
3       long[] sums = new long[n];
4
5       for (int i=0; i<n; ++i) {
6           sums[i] = sumTo(i);
7       }
8
9       long overallSum = 0;
10
11      for (int i=0; i<n; ++i) {
12          overallSum += sums[i];
13      }
14  }
15
16  long sumTo(int n) {
17      if (n == 0) return 0;
18      return n + sumTo(n-1);
19  }
```

Figure 1. The `SumUp` example program.

Tracing dynamic dependences. During the execution of a Java program, we *trace* all read and write accesses by all instructions. The traced variable accesses constitute *dynamic dependences:* If some instruction instance A reads a value that was written by an earlier instruction instance B, then B has influenced the computation at A, or A depends on B.

As an example, consider the `SumUp` program in Figure 1. `SumUp` sets each element `sums[i]` to the sum `sums[i]` $= \sum_{j=0}^{i} j$, where the sum is computed recursively in the helper method `sumTo()`. During execution, we trace (among others) that the value of `sums[1]` depends on the return value of `sumTo()`, which again depends on `n` and the return value of the recursive call. Section 2 provides foundations on how to establish dependences; Section 3 describes the implementation of our tracer.

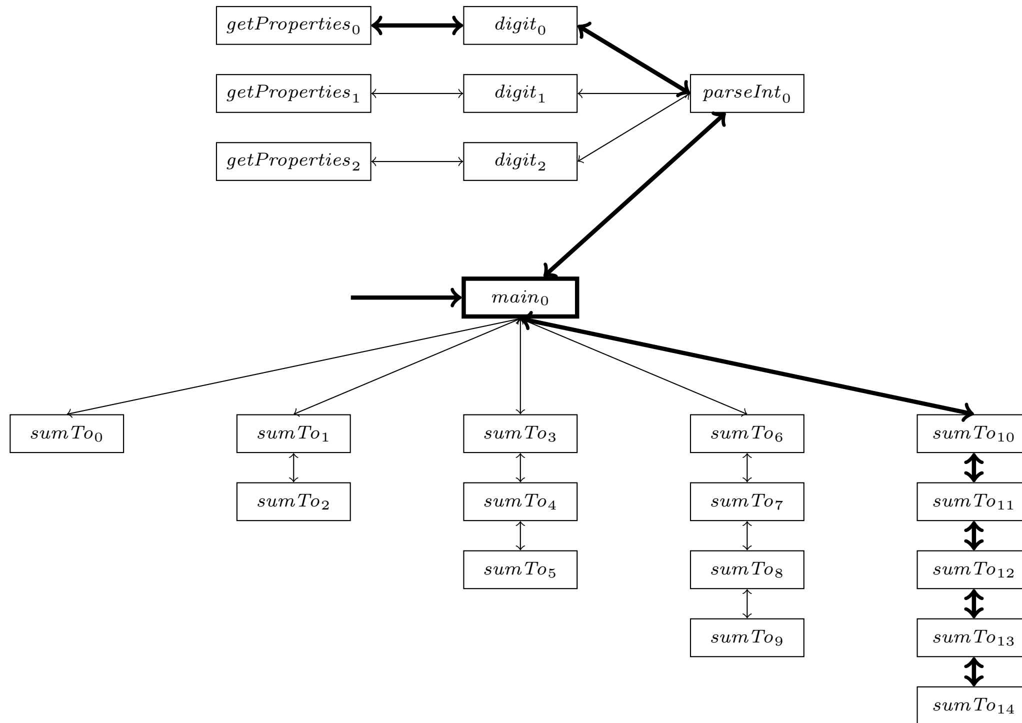

Figure 2. The method level dynamic dependence graph for input "005". The critical path (shown in bold) denotes the part of the computation that must be executed sequentially.

Detecting parallelism. The dynamic dependences form a *dynamic dependence graph* which describes how the individual instruction instances transitively depend on and influence each other. Figure 2 shows the full dependence graph of the `SumUp` program when executed with the input `005` (an input that leads to multiple digits being processed, but still a small number of `sumTo()` calls). We see how the three individual digits contribute to forming the `parseInt()` return value, which in turn triggers the invocation of five independent instances of `sumTo()`, which in turn invoke more instances. (Each invocation instance is indexed with its own serial number.)

In the dynamic dependence graph, we can identify parallel as well as serial computation paths. In particular, we can identify the *critical path*—the longest path whose instruction instances *must* be executed sequentially. In Figure 2, the critical path (shown in bold) spans from the first call to `getProperties()` (denoted as $\texttt{getProperties}_0$) to $\texttt{sumTo}_{14}$, the last invocation of `sumTo()`.

As the critical path forms a lower bound on execution time, it indicates how much a serial run can be sped up given unlimited resources for parallel execution. In our example, we could thus speed up execution down to the 9 calls on the critical path. (We compute dependences on *instructions* rather than method calls.)

In Section 4, we describe how to compute the critical path and how to compute the amount of parallelism in serial Java programs. Section 5 presents initial results on eight medium- to large-scale Java programs: In theory, these benchmarks can be sped up by factors up to 500 and more.

Suggesting parallelization candidates. Long paths of parallel computation indicate opportunities for parallelization—either *manually* (by making appropriate recommendations to the programmer) or *automatically* (by applying some form of speculative parallelization). In our example, the first loop in the `main()` function would be an obvious candidate, as all elements of the `sums[]` array are computed independently. In Section 6, we describe the approach and discuss first experiences.

To our knowledge, ours is the first tool to measure the potential for parallelism in real-life Java programs. In the long run, the approach will provide starting points for automatic parallelization of large-scale programs, as well as produce important empirical data for future research.

2. Foundations

In this section, we explain the notation and terms used throughout the rest of this paper. Readers familiar with execution traces, control-flow and data-dependence graphs may safely skip this section.

A program P is given by a set of methods $P = \{M_1, \dots\}$. Each method M is defined by its control-flow graph $G_M = (V, E, r)$. Each vertex $v \in V$ is an *instruction* of the form

$$z \leftarrow op(x_1, \dots, x_n)$$

where z is called the result of v and $x_1, \dots, x_n$ the arguments of v. The edges E model the flow of control between instructions. $r \in V$ is the root of the CFG, i.e. the instruction that has no control-flow predecessor.

Running the program P on some input D yields a new program $P(D)$ which is called the *trace* of P on D. Whenever an instruction $I : z \leftarrow op(x_1, \dots, x_n)$ in P is executed, an instruction $\hat{I} : \hat{z} \leftarrow op(\hat{x}_1, \dots, \hat{x}_n)$ is added to $\hat{P}$ and connected by a control-flow edge to the instruction added before. We also call $\hat{I}$ an *instruction instance* of I. The variables $\hat{z}, \hat{x}_1, \dots$ represent the actual addresses that were written to and read from when the instruction I was executed. Since a trace is just straight-line code, its control-flow graph is just a chain $\hat{I}_1 \to \dots \to \hat{I}_n$. Thus, we abbreviate it by the sequence of its instructions: $P(D) = \hat{I}_1, \dots, \hat{I}_n$.

In a trace $\hat{I}_1, \dots, \hat{I}_n$, an instruction $\hat{I}_k$ is data dependent on another instruction $\hat{I}_i$ if $\hat{I}_k$ uses some variable $\hat{x}$ that $\hat{I}_i$ defines and there is no $i < j < k$ such that $\hat{I}_j$ also defines $\hat{x}$. We then write $\hat{I}_i < \hat{I}_k$. The relation $<$ induces a directed, acyclic graph on the trace's instructions which we call the *dynamic data-dependence graph*. The longest path from $\hat{I}_1$ to $\hat{I}_n$ in the dynamic data-dependence graph is called the *critical path.*

3. Efficiently Tracing Programs

The basis of our parallelism-detection algorithm is the execution trace of a program run on a specific input. The *tracer* which produces these execution traces is a part of a dynamic slicer developed in our group [4]. In this section we will give a brief overview of how execution traces can be efficiently produced and stored.

The trace file is created by a *Java agent*, which hooks itself into the Java virtual machine (VM) in order to instrument every class when it is loaded by the class loader. The trace file contains all information needed to reconstruct the precise sequence of bytecode instructions that have been executed and all dynamic dependences between them. This information includes a representation of all classes that have been instrumented.

The control flow is traced in such a way that it can efficiently be reconstructed in reverse order, because all algorithms presented in this paper iterate backwards through the execution trace. For each executed instruction which has more than one predecessor in the static control-flow graph, the tracer records the dynamic predecessor. Doing the same at the beginning of each method and after each method call enables us to rebuild the precise execution trace. Special effort has been taken to handle exceptions properly.

Beside this information about the control flow, the trace file also contains dynamic information concerning the access of array elements and object fields. In order to identify the *memory address* which is read or written by an array-load, array-store, getfield or putfield instruction, each object (including arrays) is dynamically assigned an identity, which is recorded for each execution of such an instruction. Combining this information and the control flow, we compute the dynamic data dependences as described in the next section.

Currently, the tracer records all information separately for each thread, which limits us to analyze the dependences within one thread only. Because of caching effects, instruction reordering and other optimizations allowed in the Java memory model, it is much more challenging to reliably trace dependences between interleaving threads. For now, we leave this open for future research.

Since the traced sequences occurring at each single instrumented location are highly repetitive, Wang and Roychoudhoury propose to compress each of them separately [11] using the Sequitur [6] algorithm. A comparison [4] between Sequitur and gzip compression shows that gzip also performs very well (compressing to less than five percent on average), but consumes much less memory. So we are using gzip compression for all of our benchmarks.

For testing our tool set we use the *DaCapo* benchmark suite [1] which contains eleven medium to large scale Java programs three of which are multi-threaded (hsqldb, lusearch, xalan). Table 1 compares the execution time of the single-threaded programs in a traced and untraced setting and states the size of the trace file in relation to the length of the execution trace.

4. Detecting Parallelism

The amount of potential parallelism is determined by the data dependences in the program: If an instruction $\hat{I}$ consumes a value computed by an instruction $\hat{J}$ ($\hat{I}$ is data dependent on $\hat{J}$), $\hat{I}$ cannot be executed before $\hat{J}$. As mentioned above, we are interested in the amount of potential parallelism of a program P run on a particular input D. Therefore, we investigate the dynamic data-dependence graph H of the trace $P(D) = \hat{I}_1, \dots, \hat{I}_n$.

Assuming an infinite number of processors and zero cost

benchmark	description	runtime raw	runtime traced	trace length	trace file size	analysis critical path len.	analysis potential
antlr	parser generator	0.4 s	12.6 s	252,966,020	12.6 MB	733,726	344.77
bloat	bytecode-level optimization	1.5 s	29.4 s	370,075,796	21.3 MB	1,757,622	210.55
chart	graph plotting	2.4 s	20.4 s	8,553,981	7.7 MB	89,535	95.54
eclipse	Java IDE	6.6 s	103.9 s	352,435,955	174.8 MB	903,930	389.89
fop	print formatter	0.9 s	10.9 s	113,433,667	16.0 MB	493,113	230.04
jython	python interpreter	0.5 s	25.7 s	3,487,935,981	29.3 MB	15,190,620	229.61
luindex	text indexing	0.7 s	23.9 s	372,897,081	14.4 MB	647,810	575.63
pmd	Java source code analyzer	0.4 s	5.2 s	23,473,919	13.6 MB	171,593	136.80

Table 1. Results of tracing and analyzing programs from the *DaCapo* benchmark suite

of scheduling time, the instructions could be scheduled such that the trace can be computed in k steps where k is the length of the critical path in H. Thus, in this hypothetical setting, the program could be sped up by a factor of n/k. We call this number the *parallelization potential.*

Of course, real systems have a finite number of processors, and true costs for scheduling; real performance is also very much determined by locality of data accesses as well as data caching strategies. These real-life constraints can be accommodated when interpreting the data. Furthermore, the main purpose of our tool is to aid the programmer in finding code regions that could profit from parallelization. To this end, we identify regions in the trace with high potential speed up, i.e. a short critical path in the dynamic data-dependence graph.

Since the traces are very long (up to 370 millions of executed instruction in our benchmark programs), the dynamic data-dependence graphs can hardly be fully constructed in memory. Other approaches deal with this problem by using approximative methods [12]. However, we are only interested in the *length* of the critical path and therefore do not need to construct the graph in memory.

We assign each instruction instance $\hat{I}$ a length $\ell(\hat{I})$. The length $\ell(\hat{I})$ of the longest path to $\hat{I}$ is simply the maximum of the lengths of all incoming paths at $\hat{I}$ plus one, i.e. $\ell(\hat{I}) = 1 + \max_{\{\hat{J} \mid \hat{I} < \hat{J}\}} \ell(\hat{J})$. The length of the critical path can be easily computed by a single sweep over the trace:

The last instruction $\hat{I}_n$ of the trace is set to $\ell(\hat{I}_n) := 0$. Then, we traverse the trace upwards (as described above, the tracer already writes out the traces from back to front) and keep a map $live : \mathbb{N} \to \mathcal{P}(V)$. The map $live$ maps a memory address to the set of instruction instances that read that address since the last write to that address (seen from back to front). Assume we visit instruction instance $\hat{I}$ that reads addresses R and writes memory address w. The set of all nodes that are data dependent on $\hat{I}$ is just $live(w)$. Thus, $\ell(\hat{I})$ can be easily computed by adding 1 to the maximum of the lengths of the nodes in $live(w)$. Finally, we remove w's entry from the $live$ and add entries $r \mapsto \hat{I}$ for each $r \in R$.

In doing so, we visit every instruction instance only once, thus there is no need of keeping all in memory. Furthermore, the memory consumption is limited by the maximum number of simultaneously live memory addresses. Note that this can also be used on a part $\hat{I}_k, \ldots, \hat{I}_m$ of the trace: Memory addresses written without being in the live map, because they are not read with in the trace part, can simply be ignored. Thereby, potential parallelism with the outlying part is ignored. For a sequence of instructions $T = \hat{I}_1, \ldots, \hat{I}_m$ we denote the length of the critical path of T's data-dependence graph by $\ell(T)$.

5. First Results

As mentioned in Section 3, we use the single threaded programs of the *DaCapo* suite to test our algorithms. In the following we give the results of our tool set and analyze how these results come to be.

Table 1 shows the results of our tests. Additional information given are the length of one run using the option *-s small* which executes the corresponding benchmark on a small input set, the length of the critical path in this execution and the parallelization potential.

The size of the runs spans from 8,553,981 instruction instances for chart to 3,487,935,981 instances for *jython*. The parallelization potential ranges from 95.54 for chart to 575.63 for *luindex*. (As pointed out in Section 4, this number does not take overhead of scheduling into account, and should be seen as a theoretical upper bound for what non-speculative parallelization could achieve.) Generally, our results indicate a definite potential for parallelization for Java programs. They also suggest that with an increasing execution length the potential parallelism increases as well.

Where do these potential speed-up factors come from? Let us investigate what constitutes the potential for parallelization by looking at the three benchmark programs with the highest factors in more detail.

luindex is a file indexing engine. In the benchmark, five

files are indexed. The results of several files are joined at the end to form a single index. The individual indexing operations could potentially be performed in parallel as indicated by our results; the joining still is serial. This is not much of a surprise; indeed, our tool just confirmed what could be obtained from a brief code examination.

antlr is a generator for recursive descent parsers. The parser code for each rule in the grammar can be generated independently from all other rules. Therefore, all rules can be processed in parallel—a property which is not obvious from the code, and which can be exploited by programmers.

eclipse is a programming environment. The *DaCapo* benchmark runs seven JDT performance test suites. Since test suites are set up to be independent of each other, the potential for parallelization is not surprising. However, our result confirms that the tests are indeed independent—which would a very valuable information for a programmer who wants to improve testing performance.

Obviously, these first results show very coarse-grained parallelism in outer loops—which is where, at least in these programs, the most gain is to be found, and where the programs can be easily rewritten to take advantage of parallelization. However, our approach is equally applicable to detect more fine-grained parallelism.

In the long run, benchmarks created for measuring the performance of Java runtime systems may not prove best for measuring the potential for parallelization. Therefore, every result for parallelization potential needs to be checked carefully for whether the potential comes from the program—or the way it is benchmarked.

6. Suggesting Parallelization Candidates

The parallelization potential can not tell us how or where to start looking for code that can possibly be parallelized. As a first application of the dependence profiler, we investigate the potential parallelism in loops. To this end, we first compute a loop tree [7] for every control-flow graph in the program. Second, we perform the same backward traversal of the trace as for detecting the overall critical path length. Each time the execution reaches a loop boundary, a new computation of the critical path is started for this instance of the loop. Consecutive iterations of the loop body of one dynamic instance of the loop are handled as one execution for which the critical path is determined.

Consider a program P, an input D, and the resulting trace $P(D)$. Let L be a loop, given by the instructions $I_1, \dots, I_n$ of its body, and let $\hat{L} = \hat{I}_{i_1}, \dots, \hat{I}_{i_m}$ be an instance of the loop in $P(D)$. Then, the length of the critical path is determined for every instance $\hat{L}_i$ of L in $P(D)$. After processing the complete trace the parallelization potential for loop L is computed as ratio of the accumulated lengths of the critical paths and lengths of the instances:

$$\mathit{potential}(L) := \frac{\sum_{\text{Instance } \hat{L}} \ell(\hat{L})}{\sum_{\text{Instance } \hat{L}} |\hat{L}|}$$

As an example, consider the run of `SumUp` (see Figure 1) on the input “015”. The overall parallelization potential is 29.32. For the loop in line 5, the loop analysis reports a potential of 23.78, for the one in line 11 a potential of 4.02. Note that the potential of the latter loop is almost independent from the input because the loop has loop carried dependences. The potential speed-up of ~ 4 does not vary much for different inputs.

The potential alone is not very expressive, because a loop with a high potential does not influence the overall speed of the program significantly if it is not executed very often. Thus, we introduce the influence $\mathit{influence}(L)$ of a loop L. The influence is the quotient of the number of executed instruction instances of $\hat{L}$ and the length of the trace. Based on the influence and the potential L, we define the gain as

$$\mathit{gain}(L) = \mathit{influence}(L) - \frac{\mathit{influence}(L)}{\mathit{potential}(L)}$$

to couple the influence of a loop with the potential parallelism. This will direct the programmer to the loops which will benefit most from parallelization.

Applied to the *DaCapo* suite, the loop analysis pinpointed the parallelism at the right places as we verified by manual inspection. For example, in the *luindex* benchmark, almost all of the potential was found in the loop that distributes the indexing across the individual inputs.

7. Related Work

In the last years there have been several papers about extracting parallelism using dynamic analyses. However, most of the presented tools do dynamic binary instrumentation of the machine code produced by a C compiler. This causes several complications: Due to very aggressive compiler optimizations it is harder to trace the dependences. For example, the binary instrumentation must be able to trace dependences also through the register file if the compiler performed aggressive register allocation. It might even be necessary to recompile the code disabling all optimizations.

Our tool instruments Java bytecode with a Java agent. It can be used on any VM that implements the necessary API. Therefore, the program can be analyzed without modifications or recompilation. To our knowledge, our tool is the

first one that can efficiently profile medium to large scale Java programs for parallelism.

The work closest to ours is the Taskfinder tool by von Praun et al. [10]. Taskfinder is similar to our analysis framework, and works on C programs. However, the size of the programs we can profile is a magnitude larger.

Bridges et al. [2] perform automatic parallelization using decoupled software pipelining. They also find suitable regions for parallelization with a profiler that traces memory dependences. However, for safe speculation, they rely on hardware features that are not common in current systems.

Thies et al. [9] apply dynamic binary instrumentation to analyse data dependences in C programs to extract pipeline parallelism that can be used to transform the program into a stream program. The applied program transformation requires that *every* data dependence is emitted during the dynamic analysis which makes the approach unpractical for larger programs.

Rul et al. [8] presents an automatic parallelization framework for C programs that also relies on dynamic analysis. The granularity of their approach is even more coarse than in the work by von Praun et al: code between function calls and loop boundaries are compressed to so-called snippets; the dynamic data dependences are treated only between snippets.

Harris and Singh [5] use profiling to detect parallelism in functional programs written in Haskell. As Haskell is a pure functional language, there is no competing access for shared resources. Their approach aims for finding parallelism among so-called *thunks,* units of code created by the Haskell compiler to implement lazy evaluation.

None of these approaches has been demonstrated to work on object-oriented programs; or been shown to scale up to medium- to large-scale programs. This is the specific contribution of the present paper.

8. Conclusion and Future Work

By tracking the dependences between computation paths in Java program runs, our approach provides measurement for the parallelization potential of Java programs. Just like a regular profiler detects locations where most time is being spent, our approach detects those locations where parallelization brings the most benefits—an important task that can guide and should precede any kind of manual or automatic parallelization. Our early results clearly highlight the parallelization potential in real-life Java programs and thus should encourage research on how to fully exploit this potential.

Measurement alone does not bring performance benefits, though; and lots of work remain to fully leverage the power of dynamic dependences. Our future work will focus on the following topics:

Better modeling. As stated in Section 4, the amount of parallelism given by the critical path is much more a theoretical than a practical value. We want to compute realistic maximum amounts of parallelism given: a limited number of processors or other resources; a specific effort for creating threads; varying costs for different instructions; or varying costs for (possibly cached) data accesses.

A workbench for experimentation and evaluation. Dynamic dependences, as determined by our approach, can be used to easily experiment with different parallelization techniques and to evaluate them. Rather than implementing a complicated parallelization technique into some compiler, and then only evaluating its performance, one can use our traces to evaluate a concept before implementing the full tool chain.

Alternate computations. A dynamic approach like ours can also be used to detect opportunities for alternate computations. For instance, one could exploit arithmetic properties like associativity or commutativity to reorder computations and thus unlock further fine-grained parallelization potential. Similar transformations could be applied to data structures.

More and multi-threaded programs. As it comes to the implementation, our approach is currently limited to single-threaded programs. An extension to multiple threads is straight-forward, though, and will allow us to apply the approach on a larger variety of programs.

Granularity. When computing dependences, we have the choice between various levels of granularity—methods vs. basic blocks vs. instructions. We want to explore how these different levels impact the accuracy of our computations, and how they contribute to making the approach even more scalable.

Speculative parallelization. Last but not least, we want to implement a system that leverages this very measured potential by applying automated, speculative parallelization to large-scale Java programs. Before such an implementation, though, comes measurement—and the parallelization potential, as discovered in our early results, is very encouraging.

More information on this work can be found at

`http://www.st.cs.uni-saarland.de/javaslicer/`

Acknowledgments. We thank the anonymous reviewers for their detailed and constructive comments.

References

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

[2] M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August. Revisiting the sequential programming model for the multicore era. *IEEE Micro*, January 2008.

[3] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhäuser, 2000.

[4] C. Hammacher. Design and implementation of an efficient dynamic slicer for Java. Bachelor's Thesis, November 2008.

[5] T. Harris and S. Singh. Feedback directed implicit parallelism. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 251–264, New York, NY, USA, 2007. ACM.

[6] C. G. Nevill-Manning and I. H. Witten. Linear-time, incremental hierarchy inference for compression. In *Data Compression Conference, 1997. DCC '97. Proceedings*, pages 3–11, 1997.

[7] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Trans. Program. Lang. Syst.*, 24(5):455–490, 2002.

[8] S. Rul, H. Vandierendonck, and K. D. Bosschere. Extracting coarse-grain parallelism in general-purpose programs. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 281–282, Salt Lake City, Feb. 2008.

[9] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[10] C. von Praun, L. Ceze, and C. Caşcaval. Implicit parallelism with ordered transactions. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–89, New York, NY, USA, 2007. ACM.

[11] T. Wang and A. Roychoudhury. Using compressed bytecode traces for slicing Java programs. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 512–521, 2004.

[12] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *IEEE/ACM International Conference on Software Engineering*, pages 319–329, 2003.