AUTOMATIC PROGRAM PARALLELIZATION USING TRACES

by

Borys Jan Bradel

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of the Edward S. Rogers Sr. Department of
Electrical and Computer Engineering
University of Toronto

Copyright © 2010 by Borys Jan Bradel
Abstract

Automatic Program Parallelization Using Traces

Borys Jan Bradel

Doctor of Philosophy

Graduate Department of the Edward S. Rogers Sr. Department of Electrical and
Computer Engineering

University of Toronto

2010

We present a novel automatic parallelization approach that uses traces. Our approach
uses a binary representation of a program, allowing for the parallelization of programs
even if their full source code is not available. Furthermore, traces can represent both
iteration and recursion. We use hardware transactional memory (HTM) to ensure correct
execution in the presence of dependences.

We describe a parallel trace execution model that allows sequential programs to ex-
ecute in parallel. In the model, traces are identified by a trace collection system (TCS),
the program is transformed to allow the traces to execute on multiple processors, and
the traces are executed in parallel.

We present a framework with four components that, with a TCS, realizes our execu-
tion model. The grouping component groups traces into tasks to reduce overhead and
make identification of successor traces easier. The packaging component allows tasks to
execute on multiple processors. The dependence component deals with dependences on
reduction and induction variables. In addition, transactions are committed in sequen-
tial program order on an HTM system to deal with dependences that are not removed.
Finally, the scheduler assigns tasks to processors.

We create a prototype that parallelizes programs and uses an HTM simulator to deal
with dependences. To overcome the limitations of simulation, we also create another
prototype that automatically parallelizes programs on a real system. Since HTM is not used, only dependences on induction and reduction variables are handled.

We demonstrate the feasibility of our trace-based parallelization approach by performing an experimental evaluation on several recursive and loop-based Java programs. On the HTM system, the average speedup of the computational phase of the benchmarks on four processors is 2.79. On a real system, the average speedup on four processors is 1.83. Therefore, the evaluation indicates that trace-based parallelization can be used to effectively parallelize recursive and loop-based Java programs based on their binary representation.
Acknowledgements

I am grateful to everybody who has made this thesis possible. First, I would like to thank my supervisor, Professor Tarek S. Abdelrahman, who has provided me with his on-going guidance, support, and encouragement. I would also like to thank my PhD committee for providing guidance.

I would like to thank all of my family and friends for being there for me throughout this time. In particular, I would like to thank Mihai Burcea and Utku Aydonat for providing support for some of the underlying systems used in this research.

I would like to especially thank my wife, Yiqiao Wang, for her patience, understanding, support, and love during my PhD. Lastly, I would like to acknowledge the financial support provided by the National Science and Engineering Research Council of Canada.
# Contents

1 Introduction  
   1.1 Benefits and Drawbacks of Traces  
   1.2 Thesis Overview  
      1.2.1 Execution Model  
      1.2.2 Framework  
      1.2.3 Transactional Memory  
      1.2.4 Implementation  
   1.3 Experimental Evaluation  
   1.4 Thesis Contributions  
   1.5 Thesis Organization  

2 Background  
   2.1 Basic Formalisms  
   2.2 Program Representation  
   2.3 Feedback Directed Systems  
   2.4 Java Virtual Machines  
   2.5 Traces  
      2.5.1 Trace Collection  
   2.6 Transactional Memory
<table>
<thead>
<tr>
<th>3</th>
<th>The Potential of Trace-Level Parallelism</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Infrastructure</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>Simplified Parallel System</td>
<td>27</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Simplified Parallel System Simulator</td>
<td>28</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Implementation</td>
<td>35</td>
</tr>
<tr>
<td>3.3</td>
<td>Simplified Parallel System Evaluation</td>
<td>38</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Effect of the Processor Count</td>
<td>41</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Effect of the Communication Model</td>
<td>42</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Effect of the Scheduling Window</td>
<td>44</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Communication Among Traces</td>
<td>46</td>
</tr>
<tr>
<td>3.4</td>
<td>Summary</td>
<td>49</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>Parallel Trace Execution Model</th>
<th>52</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Execution Model</td>
<td>52</td>
</tr>
<tr>
<td>4.2</td>
<td>Parallel Trace Execution Example</td>
<td>56</td>
</tr>
<tr>
<td>4.3</td>
<td>Transformation Example</td>
<td>58</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5</th>
<th>Trace-Based Parallelization Framework</th>
<th>62</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Grouping</td>
<td>62</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Task Identification</td>
<td>67</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Iterative Example</td>
<td>73</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Recursive Example</td>
<td>75</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Recursive and Iterative Example</td>
<td>77</td>
</tr>
<tr>
<td>5.2</td>
<td>Extraction and Packaging</td>
<td>79</td>
</tr>
<tr>
<td>5.3</td>
<td>Dependences</td>
<td>84</td>
</tr>
<tr>
<td>5.4</td>
<td>Scheduling and Execution</td>
<td>85</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Direct Scheduling</td>
<td>86</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Queue Scheduling</td>
<td>88</td>
</tr>
</tbody>
</table>
### 6 Handling Dependences with Transactional Memory

6.1 Approach to Sequential Program Commit Order

6.2 Examples

6.3 Limits, Benefits, and Extension

### 7 Implementation

7.1 The Jikes Prototype

7.2 Grouping Component

7.2.1 Mapping Traces to the Optimizing Compiler’s Control Flow Graph

7.2.2 Identifying Edges between Traces

7.3 Packaging Component

7.3.1 Two-Part Outlining

7.3.2 Instruction Matching

7.3.3 Variable Passing

7.4 Dependence Component

7.4.1 Reductions and Inductions

7.4.2 Join Insertion

7.5 Scheduler

7.6 Hardware Transactional Memory and Java

7.7 LogTM Prototype

7.7.1 Macros for Tasks with Induction and Reduction Variables

7.7.2 Macros for Tasks without Induction and Reduction Variables

7.7.3 Setup Macros

### 8 Experimental Evaluation

8.1 Platforms

8.1.1 GEMS/Simics

8.1.2 Simulated System
8.1.3 Jikes RVM ............................................................... 137
8.2 Benchmarks ............................................................ 138
  8.2.1 Input Sizes ....................................................... 140
8.3 Methodology ........................................................... 142
8.4 Metrics ................................................................. 143
8.5 Trace Collection ..................................................... 145
8.6 Traces and Hardware Transactional Memory ................. 150
  8.6.1 Speedup of Computational Sections ....................... 151
  8.6.2 Padding ............................................................ 155
  8.6.3 Sequential Execution Time Breakdown .................... 157
  8.6.4 Dependences ..................................................... 159
8.7 Automatic Trace-Based Parallelization ...................... 173
  8.7.1 Sequential Overhead .......................................... 174
  8.7.2 Overall Parallel Performance .............................. 176
  8.7.3 Parallel Overheads ............................................ 178
  8.7.4 Importance of Scheduler Parameters ..................... 181
8.8 Comparison of Related Experimental Evaluations .......... 185

9 Related Work ............................................................. 188
  9.1 Traces ................................................................. 188
    9.1.1 Static Trace Scheduling .................................... 188
    9.1.2 Path Profiling ................................................ 189
    9.1.3 Hardware Trace Systems .................................... 191
    9.1.4 Software Trace Systems ..................................... 192
    9.1.5 Java Trace Systems .......................................... 193
    9.1.6 Traces used for Parallelization ......................... 194
  9.2 Parallelization ...................................................... 195
  9.3 Automatic Parallelization ....................................... 196
9.4 Speculation ................................................................. 198
  9.4.1 Hardware Support for Speculative Parallelization ............... 198
  9.4.2 Java Thread-Level Speculation .................................... 203
  9.4.3 Java on Hardware Transactional Memory ......................... 203

10 Conclusion ............................................................... 205
  10.1 Contributions .......................................................... 205
  10.2 Lessons Learned ...................................................... 208
  10.3 Future Work .......................................................... 210

A The Potential of Trace-Level Parallelism ................................ 214
  A.1 Simulation Methodology and Timing .................................. 214
  A.2 Complete Performance Results ....................................... 218

B Trace Collection Approaches ............................................ 222
  B.1 With Trace Exit Events .............................................. 223
  B.2 Without Trace Exit Events .......................................... 224
  B.3 Trace Collection Evaluation ......................................... 225

C Complete Timing Data for the Jikes RVM ................................ 228
  C.1 Extracting and Packaging a Trace .................................... 228
  C.2 Effects of not Grouping Traces ...................................... 229
  C.3 Tables of Results ..................................................... 230

Bibliography ................................................................. 242
List of Tables

3.1 Input set sizes for JOlden. ........................................ 39

8.1 Input and adaptive configuration execution time. .............. 141
8.2 Simulation configuration. ........................................ 141
8.3 Average cycles per good transaction. .......................... 155
8.4 Simulation configuration for dependence benchmarks. .......... 164
8.5 Number of aborts for different configurations and processor counts. ... 168

C.1 Wallclock times for BiSort. .................................... 234
C.2 Wallclock times for Perimeter. ................................ 235
C.3 Wallclock times for TreeAdd. .................................. 236
C.4 Wallclock times for TSP. ....................................... 237
C.5 Wallclock times for JGFMolDynBenchSizeA. ................... 238
C.6 Wallclock times for JGFMonteCarloBenchSizeA. ............... 239
C.7 Wallclock times for JGFEulerBenchSizeA. ..................... 240
List of Figures

2.1 Example of a simple control flow graph. .................................. 19
2.2 Traces mapped onto a control flow graph. ............................... 21
2.3 Example of trace collection. .................................................. 23

3.1 Different types of dependences. .............................................. 33
3.2 Effect of different numbers of processors. ............................... 41
3.3 Effect of the communication cost. .......................................... 42
3.4 Effect of the communication type. .......................................... 43
3.5 Effect of the trace window size. ............................................. 44
3.6 Distribution of traces. ......................................................... 47
3.7 Distribution of reads. .......................................................... 47
3.8 Distribution of reads on remote traces. ................................... 49
3.9 Distribution of traces. .......................................................... 50
3.10 Distribution of reads. .......................................................... 50
3.11 Distribution of reads on remote traces. ................................... 51

4.1 Traces executing according to the proposed execution model. ....... 57
4.2 Control flow graph transformation. ......................................... 59

5.1 Example of part of a CFG with four traces on two SCCs. ............. 66
5.2 Example of control flow between three methods. ....................... 71
5.3 Categorizing edges of an iterative program. ............................. 73
8.8 Tasks with conflicts for the recursive benchmark. ....................... 161
8.9 Tasks with conflicts for the loop benchmark. ......................... 161
8.10 Percentage of execution on transactions with conflicts for the recursive benchmark. ...................................................... 162
8.11 Percentage of execution on transactions with conflicts for the loop benchmark. ...................................................... 164
8.12 Recursive benchmark speedup. ........................................... 165
8.13 Loop benchmark speedup. ............................................... 167
8.14 Execution time breakdown for the recursive benchmark. ............. 169
8.15 Execution time breakdown for the loop benchmark. ................... 169
8.16 Distribution of execution time spent in parallel and non-parallel sections. ............................................................. 170
8.17 Speedup of \textit{rec2} benchmarks. ........................................ 171
8.18 Speedup of \textit{rec+} benchmarks. ...................................... 172
8.19 Speedup of \textit{loop2} benchmarks. ....................................... 172
8.20 Speedup of \textit{loop+} benchmarks. ..................................... 173
8.21 Time spent in different parts of execution for the default configuration on one processor. .................................................... 175
8.22 Time spent in different parts of execution when using our infrastructure on one processor. .................................................... 175
8.23 Speedup of our infrastructure relative to default configuration. ....... 177
8.24 Time spent in different parts of the main thread when benchmarks execute on four processors. ............................................. 180
8.25 Effects of varying communication, normalized with respect to the usual configuration. ..................................................... 182
8.26 Effects of varying iteration thresholds, normalized with respect to the usual configuration. .................................................. 183
8.27 Effects of varying the maximum nesting level. ......................... 184
8.28 Comparison of speedups in related work. .......................... 186

A.1 Bytecodes executed. .................................................. 216
A.2 Simulation time. ....................................................... 217
A.3 Effect of the communication type for a window of size 128. ........... 219
A.4 Effect of the communication type for a window of size 1024. .......... 220
A.5 Effect of the communication type for a window of size 8192. .......... 221

B.1 Whole program information collection. .............................. 225
B.2 Trace collection with recompilation. ............................... 227

C.1 Extracting and packaging a trace. ................................. 229
C.2 Performance on one processor. ..................................... 230
C.3 Different Jikes RVM configurations. ................................. 241
Chapter 1

Introduction

Software is used to provide solutions in a large number of problem domains. For many of these domains, the performance of the software is very important. Improved performance can lead to using less power, requiring less hardware, or completing execution in less time. Thus, fewer scarce resources need to be used to achieve the same goals. Furthermore, the improved performance can allow software to provide solutions that were not possible before.

One of the more effective and important approaches to improving software performance is parallelization. With the movement of the computer industry towards relying on multiprocessor or multicore systems to increase performance [33], parallelization is increasing in importance. However, in most cases, parallelization tends to be difficult to achieve. Therefore, a great deal of research has been done on finding ways of reducing the difficulty.

A promising research direction is automatic parallelization. There are two benefits of automatic parallelization. First, creating new software is easier due to not having to worry about parallelism. Second, existing software can also benefit from the increasing capabilities of hardware without having to be rewritten. Automatic parallelization, however, is an open research problem. Traditionally, scientific applications with regular
array accesses have been parallelized automatically with reasonable success [8, 35, 62]. However, traditional automatic parallelization has several shortcomings: it requires the source code of the entire program, it tends to work well only for programs with loops and regular array accesses, and it relies heavily on static information.

In this thesis, we pursue a novel approach to automatic parallelization: the use of traces, i.e. frequently executed paths of execution, as units of parallel work. Using traces is advantageous for several reasons: traces are a binary representation of the program, avoiding the need for source code; traces capture execution both on loops and recursive methods, allowing parallelization of programs beyond those that contain loops; and traces provide runtime information, allowing for more information to be used for parallelization.

We envision that our approach will identify traces offline and will transform a sequential program to allow its traces to execute on multiple processors. The transformation consists of identifying groups or sets of traces as coarse-grain units of parallel computation, called tasks, extracting these tasks from the original program and packaging them in a way that allows them to execute on multiple processors, and adding instructions to deal with dependences. The packaged tasks can then be scheduled at runtime to execute in parallel. This approach combines traditional automatic parallelization with the collection of runtime information in the form of traces to drive the transformations used to perform parallelization. We demonstrate the effectiveness of this approach in this thesis.

1.1 Benefits and Drawbacks of Traces

We use traces for automatic parallelization because they offer a number of potential benefits. First, traces are based on a binary representation of a program, which for this work is Java bytecode. Therefore, traces allow the automatic parallelization of programs without the need to examine source code. This is a significant advantage since in most real world scenarios the source code of the entire program is not available for examination
by a compiler. Second, traces can be parts of loops, can be parts of individual methods, or can span multiple methods. Traces may also be combined to incorporate multiple loops and methods. Therefore, traces subsume loop iterations and methods as units of parallel work, thus offering more flexibility than either loop iterations or methods and exhibiting both data and task level parallelism. Third, traces restrict the control flow that is considered for parallelization. The additional runtime information can be used to avoid spending time trying to parallelize infrequently executed sections of a program and to improve the analysis that is performed, which hopefully increases the accuracy of the analysis. Finally, traces are collected by keeping track of program execution, and are relatively simple to identify. Thus, compared to other approaches that attempt to create units of work other than loops and methods, e.g. for speculative execution [46], traces can be collected with a low overhead that will have a negligible effect on overall execution time [78].

Traces allow for the efficient collection of runtime information in a binary representation for programs that contain loop iterations as well as recursion. Thus, the successful use of traces in automatic parallelization will allow a wide variety of programs to be parallelized and take advantage of multiple processors, without access to the entire source code of the programs, without worrying about the program having certain programming constructs, and without requiring software developers to adopt new programming paradigms.

The use of traces for parallelization also has a number of drawbacks. First, collecting traces when a program executes introduces additional overhead that does not exist for traditional approaches. Second, many techniques to improve performance for existing approaches, e.g. loop tiling or loop skewing, may not be applicable to trace-based parallelization. Third, the runtime information provided by traces may be inaccurate. What the program does when the traces are collected may be different from what the program does after it is transformed (i.e. trace collection and execution are in different
phases of the program). In this case, the resulting execution may not have any improved performance. Finally, using traces is a generic approach that, without enhancement, cannot compete with domain specific approaches. For example, if an application spends practically all of its execution performing matrix multiplication, the best performance will probably be achieved by using a highly optimized linear algebra library that has its parameters tuned to the specific system executing the application. However, our research focuses on expanding the applicability of automatic parallelization instead of trying to outperform existing parallel versions of applications.

We believe that the benefits of using traces outweigh the drawbacks. Furthermore, the drawbacks can be addressed by making trace collection as efficient as possible, by future research that targets optimizations for trace-based parallelization, and by reducing the amount of time between collecting traces and transforming the program to decrease the probability of a phase change in between the two.

Further, we also perform a theoretical study to measure the amount of parallelism that exists between traces in typical benchmarks. In this study, we find that most of the benchmarks have enough trace-level parallelism to be run effectively on systems with up to four processors. Therefore, the study indicates that trace-based parallelization may lead to performance improvement and is worthwhile to research.

1.2 Thesis Overview

In this section, we give an overview of the thesis. We describe our parallel trace execution model, our parallelization framework, our approach to use transactional memory with traces, our implementation, and our experimental evaluation.
1.2.1 Execution Model

We define an execution model that provides a high level description of how a sequential program can be executed in parallel using traces. In the model, traces are identified by a trace collection system that monitors the execution of the program. The program is transformed by packaging the traces in a way that allows them to be executed in parallel. When execution reaches the start of one of these traces, execution becomes parallel. Potential successor traces are identified, and the traces are then scheduled and executed in parallel. The execution must be performed in a way that ensures that the result of the execution is the same as if the traces were executed in sequential program order.

Realizing this execution model, and successfully using traces to perform automatic parallelization poses its own challenges. We identify four major challenges.

The first challenge is to determine how to group traces together into tasks. A task is a collection of traces that form a unit of computation. Grouping is important because it allows for the reduction of overhead and for easier identification of successor traces. At a particular point in execution, the sequence of traces that will execute next has to be identified. The identification is difficult because even if traces are frequently executed, they may not be executed during certain portions of the program’s execution. Furthermore, the exact sequence of execution is hard to know a priori. In contrast to the parallelization of loops, in which all iterations are executed in parallel, determining which traces will execute requires that all control flow be known. Knowing all control flow is more difficult than knowing the number of times that a loop executes. Grouping of traces can ease this problem, because then only the control flow between tasks needs to be known ahead of time. We refer to this challenge as the grouping challenge.

The second challenge is to extract and package traces into a form that can execute in parallel. This challenge is similar to packaging loops into threads to allow them to execute in parallel. The packaging must be performed in a way that allows traces to
execute efficiently and that allows information to be passed efficiently to and from traces. However, traces are packaged at runtime and may span multiple methods, not just loop iterations. Thus, the challenge is different from that encountered when packaging loops. We refer to this challenge as the \textit{packaging} challenge.

The third challenge is to effectively schedule traces among multiple processors. This challenge is similar to the scheduling of loop iterations. However, traces are less structured than loop iterations, and therefore deciding how to schedule traces is more difficult. In particular, factors such as load balancing and data locality are more difficult to analyze. Furthermore, the identification of which traces to execute can be performed over time, and scheduling needs to take that into account. Therefore, scheduling needs to be flexible enough to handle being given new traces over time. We refer to this challenge as the \textit{scheduling} challenge.

The fourth challenge is the handling of dependences, which involves the identification of dependences and dealing with dependences between traces that are executing in parallel. An added difficulty of using traces is that control flow dependences cause early trace exits and must be dealt with. Since traces can be used to parallelize not only scientific applications with regular array accesses, but more general applications as well, determining whether there are dependences between traces will be more difficult due to more general and less uniform data access patterns. We refer to this challenge as the \textit{dependence} challenge.

\subsection{Framework}

We present a parallelization framework that addresses the above challenges. The framework takes as input traces, which are identified by a trace collection system. The framework transforms the program and then allows the traces in the program to execute in parallel. The framework consists of four components.
The first component groups traces together. Grouping consists of identifying a group or set of traces which should be executed together. Each group or set is referred to as a task. The creation of tasks has to consider two competing objectives: amortize the parallel overhead by having coarser grained tasks and allow a large amount of parallelism by having finer grained tasks. The grouping results in tasks that are coarse-grain and have predictable control flow between them. The grouping is performed by treating the traces and control flow between them as nodes and edges in a graph, and then identifying groups in the graph based on backward edges. The identification consists of finding strongly connected components (SCCs), and then subdividing them into tasks based on cycles in the SCCs. The tasks consist of sufficiently large groups of traces to keep the overhead of executing traces in parallel small. Furthermore, by creating tasks along cycles, the control flow of tasks is relatively predictable. Thus, this component addresses the grouping challenge.

The second component packages the traces. Packaging consists of modifying the instructions of a task to allow that task to be executed on an arbitrary processor or thread. The component performs steps to move the instructions in the groups of traces into their own unit of execution, adds instructions that ensure the local variables and method parameters used as inputs and outputs are passed correctly, and adds instructions that will cause the groups to be executed. The result is that the groups of traces can execute in parallel. Thus, the second component satisfies the packaging challenge.

The third component analyzes the traces to identify reduction and induction variables, and then modifies the packaged groups of traces to remove the dependences caused by these variables. However, there are other dependences that this component cannot deal with. Therefore, the component only partially satisfies the dependence challenge. To augment this component, and to fully address the dependence challenge, we use hardware transactional memory.
The fourth component schedules the groups of traces at runtime. The component can schedule in two ways. First, the groups can be scheduled by assigning them directly to processors. Second, the groups can be inserted into a queue that processors use to determine what to execute. Both are flexible and are effective at scheduling traces over time. Thus, this component satisfies the scheduling challenge.

1.2.3 Transactional Memory

We rely on hardware transactional memory to fully satisfy the dependence challenge. Hardware transactional memory allows for the efficient handling of dependences at runtime. The hardware speculatively executes multiple transactions in parallel. If there is no dependence violation, then the transactions are allowed. Otherwise, if there is a dependence violation, then the appropriate transactions are not allowed and need to re-execute. There are two main difficulties that arise from using hardware transactional memory with traces to perform parallelization.

The first difficulty is to ensure that program execution corresponds to the sequential program order of the initial program. Because we parallelize sequential programs, the parallel execution of instructions on transactions must have the same effect as the execution of the instructions in the original sequential program. However, transactional memory only requires that the transactions execute in some arbitrary order. Therefore, transactions do not have to execute in an order that corresponds to the execution of the original sequential program. We overcome this difficulty by creating a mechanism that forces an order on transactions by making transactions wait for their predecessors before committing. The work performed by the groups of traces can thus be put into transactions.

The second difficulty, which arises due to our implementation targeting Java programs, is enabling the execution of Java programs on a hardware transactional memory system. The difficulty is mainly caused by issues with synchronization. For example,
many parts of a Java Virtual Machine, such as memory allocation, interfere with transactions. Furthermore, hardware transactional memory works at a cache line granularity, which results in dependences from false aliasing. We overcome this difficulty by removing problematic behaviour, such as memory allocation and synchronization, from transactions. By using hardware transactional memory, we handle dependences efficiently, and our mechanism ensures that the execution follows sequential program order. Thus, the dependence challenge is fully satisfied.

1.2.4 Implementation

We implement an offline trace collection system to provide traces. The trace collection is performed in an initial execution, referred to as a preliminary execution, and the traces are used in a later execution, referred to as a primary execution, during which the program is transformed and executes in parallel. Collecting traces offline allows the transformation of the program to be based on all the traces that are collected and without having to worry about the efficiency of the trace collection system.

For an online system to be used, traces need to be collected and the program needs to be transformed over time as the program executes. At the beginning of execution, no traces are identified. As the program executes, more traces are identified. Finally, at the end of execution, all the traces are identified. Further, at certain points of execution, the program will be transformed. When the program is transformed, the only traces that are available to the transformation are the traces that are identified up to that point. If the transformation occurs too early, the traces that are available to it may be insufficient to transform the program effectively. Therefore, a decision algorithm needs to be created that determines when the program should be transformed.

This algorithm is non-trivial because it needs to balance two factors. The first factor is the need for transforming the program as soon as possible to extract a large amount of parallelism. The second factor is the need to delay transforming the program until
sufficient information is available to make the transformation effective. Further, the decisions by the algorithm will affect when optimization is performed, which affects the performance of the program. In addition, the implementation of the trace collection system must be efficient. Otherwise, the collection overhead can overwhelm the performance improvement from parallelization. Due to the challenges of creating an efficient online trace collection system that balances the various tradeoffs in deciding when to transform the program, we use an offline system, and leave the creation of an online system as future work.

We also implement two prototypes of our framework. Ideally, a single prototype would consist of a Java Virtual Machine (JVM) that executes on top of a real HTM system. However, that is not feasible because we do not have a real HTM system to use in our experimental evaluation. Therefore, we have implemented two prototypes of our framework, one that uses HTM and runs on a simulated system, and another that runs on a real system without using HTM. One of the prototypes uses LogTM [55] as an HTM system. The reason is that LogTM is used widely by researchers, which increases our confidence in the system being both accurate and robust. The other prototype is based on the Jikes Research Virtual Machine (RVM) [42], which is a virtual machine designed to be easily extended to perform research. The two prototypes are referred to as the LogTM and Jikes prototypes respectively. The LogTM prototype is used to evaluate the interactions between trace-based parallelization and HTM, while the Jikes prototype is used to evaluate the effectiveness and overheads of trace-based parallelization on a real system.

The LogTM prototype performs trace-based parallelization that fully handles all dependences and runs on a JVM on a simulated HTM system. The LogTM prototype parallelizes programs based on traces and uses hardware transactional memory to deal with dependences. The traces are grouped in the Jikes RVM once all the traces are collected. Macros are then used to package the groups of traces and to add instructions that
will schedule the groups. The GNU Compiler for Java (GCJ) is then used to create ex-
ecutables that can execute on the LogTM hardware transactional memory system. This
LogTM prototype has to overcome the difficulties of using HTM with traces to parallelize sequential Java programs.

The Jikes prototype runs on a real system and performs automatic trace-based par-
allelization while only handling dependences between traces based on induction and re-
duction variables. For the Jikes prototype, we add an extra phase to the Jikes RVM’s
optimizing compiler that groups traces into tasks, extracts and packages the tasks, and
analyzes and modifies the tasks to find and remove the dependences from induction and
reduction variables. Furthermore, a scheduler for the groups of traces has been added.

1.3 Experimental Evaluation

The experimental evaluation of trace-based parallelization is divided into three parts.
First, we evaluate the efficiency of trace collection. Second, we evaluate the efficiency
and effectiveness of dependence handling with hardware transactional memory. Third,
we evaluate the efficiency and effectiveness of an automatic trace-based parallelization
system. The evaluation is performed using a variety of recursive and loop-based programs
from the JOlden and Java Grande Section 3 benchmark suites. The evaluation indicates
that trace-based parallelization does extract a sufficient enough amount of parallelism for
the benchmarks to have improved performance on multiple processors. Therefore, traces
can be used effectively in combination with hardware transactional memory to parallelize programs.

We also perform an analysis to measure the amount of trace-level parallelism available
in Java programs. We define a simple parallel system that follows our parallel trace
execution model. Then, we have measured the amount of parallelism that exists for a
variety of benchmarks.
1.4 Thesis Contributions

In this thesis, we have shown that it is possible to automatically parallelize programs using only a binary representation, namely traces. Furthermore, we have shown that it is feasible to uniformly handle iteration and recursion. In this context, we make the following contributions.

- we are the first to propose, implement, and evaluate the use of traces to perform automatic coarse-grain parallelization of programs that contain both loop iteration and recursion;

- we propose an execution model for automatic trace-based parallelization;

- we create a framework that allows traces to execute according to this execution model in a way that both benefits from the advantages of traces and overcomes the challenges associated with them;

- we propose a novel solution to the issue of committing transactions in sequential program order that extends earlier work on sequential commits of transactions in loops to both loops and recursion;

- we implement a trace collection system that extends the Jikes RVM;

- we implement two offline feedback directed system prototypes that realize our framework, one prototype that parallelizes Java programs and executes them on LogTM and another prototype that automatically parallelizes Java programs and is an extension of the Jikes RVM;

- we perform a feasibility study to measure an upper bound on available parallelism among traces and create a simple simulator for this study; and
• we experimentally evaluate the performance of the two prototypes, thus substantiating our claim that traces can be used effectively in combination with hardware transactional memory to parallelize programs.

1.5 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 contains background material. Chapter 3 contains a feasibility study that examines the amount of trace-level parallelism that exists in Java programs. Chapter 4 contains a description our execution model. Chapter 5 contains a description of our trace-based parallelization framework. Chapter 6 contains a description of using transactional memory to handle dependences. Chapter 7 highlights some of the implementation details. Chapter 8 presents our experimental evaluation. Chapter 9 gives related work. Chapter 10 presents our concluding remarks and directions of future work. Appendix A contains the complete experimental data for our feasibility study and explains how the data is collected. Appendix B contains a description of our initial foray into examining different approaches to trace collection. Appendix C contains a description of extraction and packaging of a single trace and complete timing data for the results obtained using the Jikes RVM.
Chapter 2

Background

In this chapter, basic formalisms are introduced and feedback directed systems, JVMs, traces, and transactional memory are described.

2.1 Basic Formalisms

A set is an unordered collection of elements. We use formalisms that contain elements of a single type to have the formalisms correspond to the data structures provided by the Java Application Programming Interface (API). The type of a set is denoted by the type of the elements enclosed in curly brackets. Regular set operations, such as union and intersection, can be performed on sets. A specific set is denoted by a list of elements enclosed in curly brackets.

A tuple is an ordered collection of a fixed number of elements. The type of a tuple is denoted by the type of each element enclosed in parentheses. Each individual element of a tuple can optionally be given a name or identifier, which would appear after the type and can be used to refer to that element. A specific tuple is denoted by a list of elements enclosed in parentheses. A graph is a commonly used tuple that contains two sets, a set of vertices and a set of edges. Each vertex is of some specific type and is said to be an element of the graph. Each edge represents a connection between two
vertices and contains a source and target. For example, a graph of integers is defined as

\[
\text{graph} : (\{\text{int}\} V, \{(\text{int source}, \text{int target})\} E).
\]

A list or sequence is an ordered collection of elements of one type. The type of a list is denoted by the type of the elements enclosed in square brackets. A specific list is denoted by a list of elements enclosed in square brackets. Ordered lists can be appended together, which is indicated by addition. The element of a list can be referred to by a non-negative integer inside of square brackets.

A function is a relation between one group of elements to another group of elements that requires a fixed output for a given input. The type of a function is denoted by the type of the first group followed by the type of the second group with an arrow in between. A specific function is denoted by a mathematical description.

A base element is an object that corresponds to an object or datum. The type of a base element is denoted by a name and a specific base element is denoted by an identifier (variable name). Examples of base elements include integers, floating point numbers, instructions, strings, identifiers, and labels (although the last two can be thought of as strings).

A type consists of a name followed by a colon and the decomposition of the type into sets, tuples, lists, functions, base element types, and other types. Thus, more complex types can be created.

A specific entity is denoted by a type and a name, followed by an equal sign and then the entities that this entity is composed of. For a function, the name includes names for inputs. Mathematical and logic operations can be used on these inputs.

The following are some examples of types and entities for sets, tuples, lists, functions, and operations on specific entities.

- Sets

\[
\text{aset} : \{\text{int}\}
\]
Chapter 2. Background

\[
aset x = \{1, 2, 3\}
\]

\[
aset y = \{1, 2, 3, 4, 5\}
\]

\[y - x = \{4, 5\}\]

\[x \cap y = \{1, 2, 3\}\]

Above, a type of set, \(aset\), that contains integers is defined. Then two sets of type \(aset\) are created: \(x\) and \(y\). Afterwards their subtraction and intersection is shown to be equal to two sets of integers.

• Tuples

\(btuple : (\text{string, string})\)

\(btuple z = ("1", "2")\)

\(ctuple : (\text{string } a, \text{string } b)\)

\(ctuple a = ("3", "4")\)

\(a.a = "3"\)

First, a type of tuple, \(btuple\), that contains two strings is defined. Then one tuple of type \(btuple\) is created: \(z\). Then another type of tuple \(ctuple\) is defined. This tuple contains two strings that can be identified by the names \(a\) and \(b\). Then a tuple of type \(ctuple\) is created, and its first element is shown to be equal to "3".

• Lists

\(dlist : [\text{int}]\)

\(dlist b = [1, 2, 3]\)

\(dlist c = [1, 2, 3, 4]\)

\(b + c = [1, 2, 3, 1, 2, 3, 4]\)

\(b[2] = 3\)

First, a type of list, \(dlist\), of integers is defined. Then two such lists are created. Then these two lists are added and the first list’s third element is accessed.
Chapter 2. Background

- Functions

\[ \text{efunc} : \text{int} \rightarrow \text{int} \]
\[ \text{efunc} d(x) = x + x \]

A type of function is defined that takes as input an integer and outputs an integer. Then a function of that type is defined. The function adds the input to itself.

2.2 Program Representation

A computer program is a sequence of instructions that can execute on a computer. These instructions tell the computer how to store and process information. The instructions can be represented as static instructions or dynamic instructions. Static instructions represent the computer program as it is stored in a computer’s memory. Usually, these static instructions are generated by a compiler based on source code that contains functions or methods written in a high level language. The computer performs operations according to these static instructions. Dynamic instructions are the stream of operations that the computer performs. Instructions can be categorized into arithmetic, memory, and control flow instructions. Arithmetic instructions perform arithmetic operations such as addition or subtraction, memory instructions move information between different locations in memory, and control flow instructions specify which static instruction to execute subsequent to the control flow instruction. Since a computer can execute multiple operations in parallel, there can be multiple streams of dynamic instructions, with the instructions that occur at the same points in the streams executing in parallel. We refer to each stream as an execution context. All the dynamic instructions from the different execution contexts can be represented as a tuple of sequences, one per execution context:

\[ \text{execution} : ((\text{instruction}), (\text{instruction}), \ldots, (\text{instruction})) \].

This representation is abstract and does not capture architectural details such as processors, pipelines, processing units, out of order execution, and the memory system. The abstraction allows for a concise representation of executing a program in parallel without having to simultaneously address these other, and mostly orthogonal, issues.
A program’s static instructions can be represented by a control flow graph (CFG). The advantages of a CFG are that it is amenable to transformation and can be easily shown graphically. A CFG contains a set of vertices and a set of edges. Each vertex is a basic block. A basic block consists of a label and a sequence of instructions. A basic block may be represented just by its label to ease readability. The most common definition, which we use without considering exceptions, is that a basic block is the largest sequence of instructions that has one entry and one exit point. Each edge is an ordered tuple that contains a source vertex and target vertex. The edges represent control flow between basic blocks. The following describes a CFG and basic block:

\[
CFG : (\{\text{basic-block}\} V, \{(\text{basic-block source}, \text{basic-block target})\} E),
\]

\[
\text{basic-block} : (\text{label label}, [\text{instruction} \text{instructions}]^2).
\]

A large amount of detail is not shown in CFGs, such as the underlying register and memory allocations for variables. Although these details must be handled by a compiler, they would only complicate the high level description, and are therefore omitted.

Because traces can span multiple methods, the CFGs that we use need to consider multiple methods. We refer to these CFGs as interprocedural CFGs [38]. Since interprocedural CFGs consider control flow between methods, call instructions will end basic blocks on these CFGs. In contrast, the traditional definition of CFGs only considers a single method [57]. We refer to such CFGs as intraprocedural CFGs. Intraprocedural CFGs are only discussed in Section 7.2, when dealing with the implementation of the optimizing compiler of the Jikes RVM.

Since each execution context needs to have a first and last instruction to execute, these need to be indicated on the CFG. The basic block that has the first instruction for execution context \( n \) has a label that ends in “\text{START-}\text{Pn}” and the basic block that has the last instruction for execution context \( n \) has a label that ends in “\text{END-}\text{Pn}”. For a program with one execution context, such basic blocks can be omitted given that the

\[\text{The type of the label element is also label. Thus, the word label appears twice.}\]
execution flow can be identified without them. A CFG for a single execution context is referred to as \textit{sequential} and a CFG for multiple execution contexts is referred to as \textit{parallel}.

For example, consider a computer with two execution contexts and a program with the CFG shown in Figure 2.1. The CFG contains four basic blocks. Each basic block has its label on the left and its instructions inside. The execution would be \( \text{execution } e = ([\text{START-PO, END-PO}], [\text{START-P1, END-P1}]). \)

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{example CFG}
\caption{Example of a simple control flow graph.}
\end{figure}

\section{Feedback Directed Systems}

A feedback directed system monitors the execution of a program and collects information that is used to optimize the program. An offline feedback directed system requires that the program is executed twice. We will refer to the two executions as \textit{preliminary} and \textit{primary}. The preliminary execution is the one in which the information is collected. After the preliminary execution, \textit{optimization}, which in this thesis is parallelization, uses this information to modify the program. The primary execution is the execution of the program after it is optimized.

An online feedback directed system requires only a single execution of a program because the system collects information and optimizes the program as the program executes. Both offline and online systems have their advantages. In online systems the feedback loop is shorter, because the collected information is immediately used while a program is
executing. Offline systems can analyze collected information more thoroughly since they
do not compete with the executing program for resources. The additional analysis may
lead to more effective optimizations.

2.4 Java Virtual Machines

Java programs are composed of methods that contain bytecode instructions. A Java Vir-
tual Machine (JVM) executes a program by either interpreting the bytecode instructions
in the methods or compiling and then executing the methods. High performance JVMs
use a Just-In-Time compiler to optimize only frequently executed methods because the
overhead of optimizing less frequently executed methods at runtime outweighs the benefit
of executing the optimized version of these methods. Such a JVM uses information about
how often methods are executed to choose the methods to optimize. Given the previous
definition of a program, a program with source code written in Java that is compiled
into bytecode has a very large CFG that consists of everything that the JVM will do
along with the compiled bytecode. We focus only on that part of CFG that represents
the compiled bytecode, and not the part that represents what the JVM will do. Therefore,
such actions as garbage allocation and context switching are not included, and if
need to be shown, will be represented by a basic block with an appropriate label and no
instructions.

2.5 Traces

A trace is a sequence of \( n \) unique basic blocks, \([b_1, b_2, \ldots, b_n]\), such that basic blocks
\( b_1, b_2, \ldots, b_n \) are executed in sequential order during the execution of a program [4].
Block \( b_1 \) is called the start of the trace and \( b_n \) is called the end of the trace. The trace
may contain any basic blocks of the program as long as the sequence corresponds to a
path on the control flow graph. Since a trace can contain multiple basic blocks it can
contain multiple points at which control flow can exit the trace. These points are referred to as *trace exits*.

An example of traces is in Figure 2.2, which contains two copies of a control flow graph, each with a different valid trace. The traces are \([B_0, B_2, B_1]\) and \([B_1, B_2]\). In contrast, the sequence \([B_1, B_2, B_1]\) is not a valid trace because \(B_1\) appears twice\(^3\).

![Figure 2.2: Traces mapped onto a control flow graph.](image)

Traces are generated by a trace collection system (TCS) that monitors a program’s execution and collects traces based on this execution [5]. The TCS starts recording a trace when occurrences of certain events exceed a specific threshold. The recording stops when certain events occur. Once the traces are collected, they can be used for optimization.

### 2.5.1 Trace Collection

Traces are generated by a trace collection system (TCS) that monitors a program’s execution and collects traces based on this execution. The TCS can start recording a trace when occurrences of certain events exceed a specific threshold. These events include a *backward*\(^4\) taken branch, a *backward taken jump*, a *return*, an *invoke*, and a *trace*

---

\(^3\)Code duplication can be used to avoid this limitation, at the expense of increasing code size.

\(^4\)A control flow instruction that goes to an earlier instruction in code order (e.g. with a lower bytecode index) is referred to as backward.
exit. We use all these events except for returns to start traces. The events allow traces to capture execution that encompasses both loops and recursion. Because the criteria to start recording traces affects which traces are created, and therefore parallelized, it may be made complex to increase parallelism (e.g. collect dependence information and only collect traces when the lack of dependences meets certain criteria). However, more complexity leads to larger overheads. Thus, the criteria has to also be simple enough for trace collection to have a reasonable overhead. The events we have chosen can be measured efficiently and correspond to natural control flow boundaries of programs.

The recording can stop when a backward taken branch or jump, an invoke, a return, or the start of a recorded trace is reached. Recording can also stop when the block that is about to be recorded is part of the current trace, or a trace becomes too long. The control events that we use for stopping recording are backward branches, and reaching the start of either a recorded trace or the trace being recorded. Once recording stops, the newly formed trace is stored in a buffer referred to as a trace cache, and the resulting traces can be used in the next step of the framework\textsuperscript{5}.

An example of the operation of a TCS is shown in Figure 2.3, which has JVM information on the left and TCS information on the right. The JVM contains the program to execute as well as storage for the program’s variables. These variables’ values are modified as shown when the program is executed. The lower part of the figure shows the sequence of steps that the JVM performs when executing the program.

The TCS contains three components: a set of event counters that are used to determine when recording should start, a recording buffer that is used to hold basic blocks as they are recorded, and a trace cache. The TCS knows when the control flow instructions are executed and acts upon the knowledge. In the figure, the left side of the TCS contains the actions of the TCS. Solid arrows show the link between individual control flow in-

\textsuperscript{5}We take these starting and stopping criteria as a given. We leave exploring different criteria, and the effects they make have on parallelism between traces for future work.
instructions and the resulting actions in the TCS. The dotted arrows indicate the recording of a trace. Within the TCS, solid arrows show the updating of the TCS’s components.

In Figure 2.3 the JVM executes basic blocks B1 and B2 repeatedly and the TCS keeps track of how often the backward branch between B2 and B1 is taken. When the backward branch is taken often enough\(^6\), the TCS starts recording a sequence of basic blocks (i.e. a trace). The TCS records the execution of the program until it detects that the next basic block is already in the recorded sequence. After recording is stopped the sequence of basic blocks, \([B1, B2]\), is stored in the trace cache. After the trace \([B1, B2]\) is saved, instruction i3 is executed and the system detects that i3 is in B1, which is the head of Trace 1. The TCS keeps track of this trace’s execution until the loop exits. This is shown by incrementing the number of times that the trace starts.

\(^6\)We have set the threshold for the counter to 2 for this example.
2.6 Transactional Memory

Transactional memory is a programming model that ensures that the execution of multiple instructions inside of the same section, called a transaction, results in behaviour that is equivalent to all the instructions in that section executing atomically [55, 79]. Transactional memory systems achieve their goal by keeping track of memory reads and writes, and then dealing with dependences that occur, either by stalling a transaction or by aborting and re-executing it. Both hardware and software transactional memory systems exist. Hardware transactional memory systems have lower overheads since reads and writes are tracked in hardware. Hardware systems are also easier to use because programs need to only identify the transactions. Software systems must keep track of reads and writes in software, thus incurring significant overheads. Further, all memory reads and writes must be modified to interface with the software system. The main advantage of software transactional systems is that they can be used on existing widely available hardware.

Each transaction is delineated with transaction begin and transaction commit instructions. Also, certain special instructions can be executed without the transactional memory system keeping track of reads and writes. Such instructions are said to escape the transaction, and the region that contains them is delineated with begin escape and end escape instructions\(^7\). Furthermore, a transaction abort represents a transaction being aborted, which occurs when a dependence has been violated. [55, 79]

For a transactional memory system, two behaviours need to be specified. They are version management and conflict detection, and they can be either lazy or eager. Eager version management places a newly stored value directly in the memory location being stored to, while lazy version management places a newly stored value into a buffer and will write to the memory location at a later point in time. Eager conflict detection

\(^7\)We use escape instructions to communicate between transactions without dependence violations. This communication allows us to order transaction commits. The details are in Chapter 6.
detects dependence violations when the stores and loads are performed, while lazy conflict
detection detects the violations at a later point in time. [55, 79]
Chapter 3

The Potential of Trace-Level Parallelism

We have performed a feasibility study that examines the amount of trace-level parallelism that exists in Java programs [11]. More specifically, we extend an existing Java Virtual Machine (JVM), Jupiter [27], that executes programs sequentially by allowing it to simulate the parallel execution of traces. The simulated execution is used to compute the amount of parallelism. We consider the effects of factors such as the number of processors, interprocessor communication cost, and communication type on parallel performance. Finally, we characterize the behaviour of traces based on where they read data from. The results indicate that most of the benchmarks have enough trace-level parallelism to result in good performance on systems with up to four processors, with some benchmarks being able to scale to eight processors. The results also indicate that for the remaining benchmarks the main source of such poor performance is the lack of parallelism as opposed to high interprocessor communication costs. Therefore, we conclude that exploiting trace-level parallelism is a viable approach to improve the performance of Java programs.
The remainder of this chapter is organized as follows. Section 3.1 contains a description of the underlying infrastructure. Section 3.2 describes our abstract parallel system. Section 3.3 shows our experimental evaluation. Finally, Section 3.4 contains a summary of the study.

### 3.1 Infrastructure

The underlying infrastructure used to identify traces consists of Jupiter [27], an interpreter based JVM, and RedSpot [9], a trace collection system that is an extension of Jupiter. Jupiter’s interpreter performs three distinct tasks for each interpreted bytecode instruction: the bytecode is executed by updating the appropriate data structures in Jupiter, the next bytecode to be executed is identified, and Jupiter’s state is updated to enable the interpretation of the next bytecode. Also, a fourth task is performed for all bytecodes that represent control flow instructions: RedSpot is called to allow it to identify the basic blocks and traces executed by Jupiter based on the current and next bytecode.

RedSpot counts the number of times that certain events occur. Each event is the encounter of a specific return, trace exit, or backward taken branch or jump. When the number of times that an event is encountered reaches a certain recording threshold, recording of a trace starts. The recording of the trace stops when a backward branch or jump or the start of another trace is encountered [9].

### 3.2 Simplified Parallel System

An important question to address is: *how much parallelism exists when applications are executed with their traces being allowed to execute in parallel.* This question is best answered by simulating the execution of several applications on an abstract parallel system that allows traces to execute in parallel. The results are not an indication of the
performance of a real system. Rather, they are an indication of the amount of parallelism that exists in applications, which is what this study is concerned with.

The advantage is that an approximate upper bound can be found quickly\(^1\). The disadvantage is that simplifying assumptions are made that may result in more parallelism than can be realized in actual hardware systems. We believe that such assumptions are acceptable because we are measuring an upper bound on performance. How to efficiently exploit the parallelism that exists requires the creation of a system that can run on existing hardware. Such a system is more complex and time consuming to create.

### 3.2.1 Simplified Parallel System Simulator

The simulator keeps track of a program’s Java bytecodes and the memory locations that are accessed as the bytecodes are executed by a JVM. Instructions that are not on a trace are executed sequentially. Instructions on traces are executed in parallel, and scheduled in a way that enforces the data dependences that exist among the traces.

The simulation measures the total number of cycles that are required to execute an application in parallel relative to executing the application sequentially. These summary scheduling results are used to compute an *abstract speedup* of the system. This metric is used to measure the trace-level parallelism that exists in Java programs.

Although the abstract speedup is based on a single execution of a program with a specific input set, the evaluation in Section 3.3 is based on benchmarks with input sets that make the benchmark execute in a representative fashion. Therefore, the evaluation is sound.

The scheduler takes communication of data into account. When reads and the writes that these reads depend on are on different processors, the scheduler ensures that each read is executed after the data is transmitted to the read’s processor. Therefore, when

\(^1\)The upper bound is approximate because it is based on the specific abstract parallel system being simulated. The design and parameters of the system affect the measurement, and there is a possibility that a real system could have a design and parameters that yield a higher speedup.
such reads are executed, the correct data is always available for them to use, and no explicit synchronization primitives need to be inserted on the traces to ensure that the data being read is correct.

The maximum number of traces that can be scheduled together is referred to as the trace window size. At the end of each trace window, the scheduler is called to schedule the traces in that window on the abstract system. Scheduling is performed with precise knowledge of which traces execute in each window. This information is obtained from the JVM. Therefore, control flow is predicted correctly within each window. However, no information is kept between traces in different windows and such traces cannot be scheduled to execute in parallel. Thus, the size of the trace window limits parallelism.

There are advantages and disadvantages of different window sizes. The advantage of a large window size is that the maximum possible parallelism can be achieved. The disadvantage is that the simulated performance does not reflect the behaviour of a real system. The system assumes that all the traces in each window are going to execute. This assumption is valid for the simulator. However, in a real system, prediction needs to identify which traces are going to execute. Since the prediction cannot always be correct, the more traces need to be predicted, the lower the probability of correctly predicting all of them, and the higher the overhead cost of dealing with the incorrect predictions\textsuperscript{2}. Therefore, performance on a real system would degrade as the number of traces increases. Thus, a large window size causes the simulation to diverge from the behaviour of a real system. The advantage of a smaller window size is that it makes the simulation more realistic and the disadvantage is that less parallelism can be identified. A good choice for a window size is one that allows a reasonable amount of parallelism to be identified and contains few enough traces that a real system could predict the execution of the traces with a high level of probability.

\textsuperscript{2}Studying the effects of different prediction heuristics and misprediction costs is left as future work.
The remainder of this section contains a description of our parallel system model. Although the model is too simplistic to precisely capture a realistic parallel system, the model captures sufficient details to give an indication of the parallelism that exists among traces.

**Processor Model**

The abstract system contains a fixed number of processors. Each processor executes instructions sequentially, in order, and one at a time. All data is in memory and there are no registers. The memory consists of both the heap and the stack. This model corresponds to the way that Java bytecodes access data without using registers. The number of cycles that a processor takes to execute an instruction is equal to the number of words the instruction writes to memory. Most instructions will take one cycle to execute, which is the ideal behaviour of a pipelined processor that can complete one instruction per cycle. Since control flow instructions do not write data, they take zero cycles to execute. Instructions that write multiple words either deal with floating point values or are sequences of simple instructions that are combined for conciseness. Thus, such instructions should take longer to execute, and this behaviour is captured by making the number of cycles that an instruction executes be equal to the number of words that the instruction writes.

The memory is shared between all processors and data can be read directly from memory without incurring additional delays. However, if data is read on one processor when it was written before on another processor, then the read must wait until the data is transmitted between the processors. This behaviour is similar to having the working set fit in a cache and having to miss in the cache when another processor invalidates the data.

This approach to modeling performance is not standard. The standard approach is to have a model based on instructions that operate using registers, memory, and functional
units. Since we are focusing on Java and using a JVM that executes Java bytecodes, we take a different approach. Our abstract processor model is based directly on Java bytecode instructions.

Further, we have chosen to base the amount of time each bytecode instruction takes on the number of words that the instruction writes. In a real system, the number of cycles taken by each instruction depends not only on what the instruction does, but also on the effects of the preceding instructions on the state of the system (e.g. the contents of caches and branch prediction buffers). In our approach, a list scheduling algorithm is used (see Section 3.2.2). This algorithm assumes that, except for communication between processors, the amount of time between instructions is fixed. The simplest and most uniform approach to identifying the number of cycles that a bytecode instruction takes is to use the number of words written. Although this approach is optimistic, we believe it is sufficient. The reason is that we are not measuring the absolute performance of the system. Instead, we are measuring the effect on speedup of increasing the numbers of processors used to execute software.

The representativeness of the speedup obtained from the abstract system depends on the independence of the speedup obtained by executing a stream of instructions from the number of cycles required to execute individual instructions. Although the two are not completely independent\(^3\), we believe that they are sufficiently independent to allow us to measure an approximate upper bound on parallelism using our abstract system.

\(^3\)For example, instructions that need to execute sequentially could be slower on a real system relative to those that are executed in parallel. Alternatively, some instructions could be slower if executed in parallel instead of sequentially. Both scenarios would reduce speedup on a real system, although they would not be captured in the simulation of our abstract system.
Communication Model

Communication between processors occurs when an instruction on one processor writes a value that is read by an instruction on another processor. Two factors affect when data that an instruction depends on is ready for that instruction’s use.

The first is whether the data is written and read on the same processor or not. If the read and write are on different processors, then communication cost is incurred. The cost has two components: a fixed cost for all communication between a pair of traces on separate processors, and a cost that is proportional to the number of words being transmitted. The fixed communication cost is defined as the number of cycles that need to elapse between two communicating traces on separate processors. The variable communication cost is defined as the number of cycles that need to elapse for each word that is transmitted between the two traces. The communication cost between any two traces is the fixed communication cost plus the number of reads times the variable communication cost.

The second factor is whether data is shared between the beginnings and ends of traces or instructions. Four possible combinations exist:

- the data is ready at the end of the write instruction and can be read immediately;
- the data is ready at the end of the trace that the write is on, and the data needs to be read at the beginning of the trace that the read is on;
- the data is ready at the end of the write instruction, and the data needs to be read at the beginning of the trace that the read is on; and
- the data is ready at the end of the trace that the write is on and can be read immediately.

Each of these is referred to as instruction–instruction, trace–trace, instruction–trace, and trace–instruction communication, respectively. The different types of communication impact the amount of parallelism because they control the amount of overlap between
communication and execution. These combinations are shown in Figure 3.1, which contains two traces, one writing \( x \) and one reading \( x \).

![Figure 3.1: Different types of dependences.](image)

Preserving all the dependences between reads and writes would under-report the amount of parallelism available. The reason is that some instructions modify data in an inherently sequential manner, even though this data can be precomputed. An example of such behaviour is the updating of induction variables. Therefore, if other instructions depend on the data written by such special instructions, this dependence must be removed. All instructions that perform a read and a write from the same location are classified as \textit{induction instructions} (e.g. \( i = i + 1 \)), and assume that any subsequent instruction that reads the produced data will be dependent on the instruction that initialized the data before the induction occurred. Thus, the dependences caused by these instructions are removed. This classification identifies the primary induction variables in all the benchmarks that are used in the experimental evaluation. Although removing more dependences may lead to more parallelism, taking into account all possible ways
of removing dependences is not feasible. Removing more dependences is left as future work. Removing dependences will only improve the amount of parallelism available, thus enabling more parallelism, and making traces a more viable parallelization approach.

**Dependence Characterization**

The dependences between traces can be classified based on the relative location of the reads and writes that cause dependences. A trace can read data from three different sources\(^4\). The first is data produced by the trace itself. The second is data produced by another trace scheduled on the same processor. Finally, the third source is data produced by another trace that executed on a different processor. Reads that use these three different sources respectively are referred to as *on-trace reads*, *on-processor reads*, and *off-processor reads*. Only the last type incurs an overhead of communication between processors. This classification can be performed only after scheduling is performed. It is only after scheduling that on-processor and off-processor reads can be distinguished.

Traces are classified according to three groups: those that contain some off-processor reads, those that contain some on-processor reads but no off-processor reads, and those that only contain on-trace reads. The traces in these groups are referred to as *remote*, *local*, and *isolated* traces respectively. Examining each group of traces separately allows a more detailed analysis of the sources of parallel overhead.

The off-processor reads are further subdivided into *critical reads* and *non-critical reads*. A critical read is an off-processor read on a trace that has been scheduled close in time to the trace that generates the data being read. Close in time is defined as being within the communication cost between processors. These reads are critical because any delay in communication will hinder performance. The non-critical reads are off-processor reads on a trace that has been scheduled far apart in time to the trace that generates

\(^4\)Reads of constants are ignored since these values can be known ahead of time and do not cause dependences.
the data being read. These reads have less of an impact on performance because even though they incur communication overhead, this overhead can be hidden by executing useful work.

The above classification allows us to reason about how dependences influence performance. An application that has many isolated traces should have a large amount of parallelism. Similarly, an application with many remote traces indicates that although data dependences exist, it is possible to schedule traces on multiple processors. Nonetheless, the presence of these dependences may affect performance due to communication overhead, especially when a large number of critical reads exists. In contrast, an application that has a majority of local traces will probably have little parallelism. The reason is that local traces are traces that had to be scheduled on the same processor with the traces that they depend on, thus causing serialization. Conversely, a small number of local traces and reads on local traces should indicate that a large amount of parallelism exists. Therefore, there are two likely sources of performance loss. The first is communication overhead incurred by critical reads. The second is lack of parallelism because of serialization, which is indicated by large numbers of local traces. The impact of these sources is examined in Section 3.3.4.

### 3.2.2 Implementation

We have created a simulator that extends Jupiter [27] and RedSpot [9] to measure trace-level parallelism. This simulator is called PIE, which stands for Parallelism Identification Engine. PIE receives information from Jupiter and RedSpot, generates scheduling information one trace window at a time, and aggregates the results to produce a measure of parallelism.

PIE is informed about the instructions that execute, the memory that these instructions read and write, and the traces that these instructions are on. This information is used to keep track of dependences. The dependences are between each memory location
and the write that modifies that location at a given moment in time and between reads and writes. The dependences are used to schedule the traces on the abstract parallel system.

PIE is given this information in the form of events from Jupiter and RedSpot. The events are generated as the instructions are executed sequentially by Jupiter. Thus the traces are generated based on a sequential execution of the program and there is no scheduler which could affect the traces. At the beginning of each instruction, an event is sent that states which instruction executes and what trace, if any, it is on. When a memory location is read or written, an event is sent that states which memory location was accessed and whether it was read or written. Since Java bytecodes access all data through either the stack or the heap, all data has a memory location associated with it. PIE stores these events in a queue.

The queue of events is processed when PIE is informed about the execution of control flow instructions. A queue is used because the trace that an instruction is on is not always known when that instruction is executed. The reason for this behaviour is that RedSpot processes information only at control flow instructions, and uses this information to determine not only which trace the next instructions will be on, but in certain cases, which trace the instructions before the control flow instruction are on. Therefore, PIE can only process instructions in the event queue once a control flow instruction is in the queue ahead of them. When PIE is sent an event that corresponds to a control flow instruction, it processes the events in the queue and correctly assigns each instruction to an appropriate trace.

PIE uses a hash table to associate each memory location with a write and a graph to represent dependent traces. When a write is processed, the memory location being written is associated with the write and its trace. When a read is processed, the write and its trace which are associated with the read location are identified. If the write and read are on the same trace or the write wrote a constant or an induction variable, then the
read is assumed to be independent of the write. The reason is that the written values can be identified ahead of time. Furthermore, the number of reads that are on the same trace as writes is recorded. For all other writes, a dependence is created between the trace that performs the write and the trace that performs the read. Anti and output dependences are not an issue because the data is written as the instructions are executed by Jupiter, and is therefore written in the order of the sequential execution of the instructions. Thus no aliasing occurs.

Once enough traces execute, the dependence graph is given to a scheduler. PIE uses the modified critical-path (MCP) algorithm, which is a list scheduling algorithm that takes communication costs into consideration [76]. The algorithm calculates the schedule of a dependence graph of traces.

Each node on the graph is a trace and its weight is the number of writes on the trace, which are assumed to execute sequentially and in order. The traces are scheduled in a way that ensures that all reads occur far enough ahead of writes and therefore no extra stalls will occur.

Each edge on the graph corresponds to a dependence between two traces. The weight of the edge is computed based on the number of writes and the type of communication that is used (e.g. trace–trace or instruction–instruction). If the type of communication involves instructions, then the read-write pair that causes the largest overhead for instruction–instruction communication is used. The weight is the sum of three values: a fixed communication cost, a communication cost proportional to the number of read-write pairs between the traces, and the number of cycles that need to exist between the start and end of the trace based on the communication type.

The schedule generated for the graph indicates the speedup obtained by using multiple processors, and the aggregate result is presented in Sections 3.3.1–3.3.3.

After scheduling, the dependence graph is traversed. Each dependence is categorized based on the processor that the traces that perform the read and write are on and how
many cycles elapse between the traces. This information is then aggregated and combined
with the information about how many reads are on the same traces as the writes that
they depend on. The aggregated information is presented in Section 3.3.4, where it is
used to analyze the impact that dependences have on performance.

3.3 Simplified Parallel System Evaluation

The simulator, PIE, is an extension of Jupiter [27] with RedSpot [9] and uses MCP [75, 76]
for scheduling. RedSpot uses a threshold of 42 to start recording traces (see Section 2.5).
This threshold has been shown in prior work to yield traces with good characteristics [9].
GNU Compiler Collection (GCC) 3.1.1 and the Blackdown Java Development Kit (JDK)
1.4.2 are used for compilation. The experiments are run on a Socket 954 Athlon 64 3000+
processor with 512 MB of RAM with Ubuntu Linux 6.10.

The amount of available parallelism is measured for sequential applications from the
Java Grande benchmark suite [44], the SPECjvm98 benchmark suite [68], and the JOlden
benchmark suite [15, 47]. The Java Grande applications are moldyn, raytracer, euler,
montecarlo, and search. The SPECjvm98 applications are compress, jess, db, javac, mpeg-
gaudio, and jack. The JOlden applications are bh, bisort, em3d, health, mst, perimeter,
power, treeadd, tsp, and voronoi. Because a simple simulator is used and no transforma-
tions of the benchmarks is needed for this evaluation, an extensive set of benchmarks can
be used. When the benchmarks need to be transformed and a real system or complex
simulator is used, the resulting complexities would make using such an extensive set very
difficult.

The benchmark suites represent three different types of applications. Java Grande
contains scientific floating point array-based programs. SPECjvm98 contains mostly
general purpose integer programs with some array based applications. Finally, JOlden
contains programs that extensively use dynamic data structures such as linked lists and
trees. Thus, the selection of benchmark suites represents a good cross section of Java applications.

Two of the benchmarks are slightly modified to avoid limitations of Jupiter. Jess has been modified to make its inner classes explicitly stated\(^5\) and voronoi has been modified to avoid a division by zero\(^6\) in the `createPoints` method of the `Vertex` class. Jupiter would fail to execute these benchmarks without the changes.

The smaller of two input data sets is used for the Java Grande suite. Using the larger input data sets results in similar simulated performance and therefore showing graphs for the larger set would be redundant. The default (large) input data set is used for the benchmarks in the SPECjvm98 suite. The JOlden benchmarks must have their input data set sizes set individually. The sizes are shown in Table 3.1.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>bh</td>
<td>1024 bodies</td>
</tr>
<tr>
<td>bisort</td>
<td>150,000 elements</td>
</tr>
<tr>
<td>em3d</td>
<td>2000 nodes, 100 degrees</td>
</tr>
<tr>
<td>health</td>
<td>5 levels, 500 iterations</td>
</tr>
<tr>
<td>mst</td>
<td>750 nodes</td>
</tr>
<tr>
<td>perimeter</td>
<td>16 levels</td>
</tr>
<tr>
<td>power</td>
<td>fixed size</td>
</tr>
<tr>
<td>treeadd</td>
<td>20 levels</td>
</tr>
<tr>
<td>tsp</td>
<td>17,000 cities</td>
</tr>
<tr>
<td>voronoi</td>
<td>20,000 points</td>
</tr>
</tbody>
</table>

Table 3.1: Input set sizes for JOlden.

The first \(2^{27}\) instructions are executed without collecting statistics and the subsequent \(2^{25}\) instructions are executed while statistics are collected. PIE exhibits excessive memory use for two benchmarks: `mst` and `treeadd`. This behaviour limits the size of the inputs that can be used. The consequence is that less than \(2^{27}\) instructions are executed by these benchmarks. Therefore, instead of \(2^{27}\) instructions, \(2^{26}\) instructions are executed for `mst`.

\(^5\)Although in Java an inner class can be referred to without being defined in the source code, Jupiter requires an explicit class definition (e.g. `class InnerClass { InnerClass() { return; } }`) for the inner class.

\(^6\)The division is of a double by an integer. Although other JVMs can handle this case by using floating point arithmetic, Jupiter throws an exception that causes it to crash.
and $2^{24}$ instructions for \textit{treeadd} without collecting statistics. The number of instructions executed when statistics are collected does not change. These settings allow us to capture the steady state behaviour of the benchmarks\textsuperscript{7}.

The default configuration is an abstract system with four processors, a window size of 1024 traces, and trace–trace communication with zero variable cost and a 20 cycle fixed communication cost. The effect that these parameters have on parallelism is examined by varying them and measuring the resulting parallelism.

The metric used to measure the amount of parallelism of each application is \textit{abstract speedup}. The abstract speedup is defined as the ratio of the number of cycles that the sequential execution of an application takes to the number of cycles that the parallel execution of an application takes to execute on the abstract system with multiple processors. Since an upper bound of parallelism is measured, the overhead of identifying and scheduling traces is not considered.

For each application, three sets of experiments are performed. In the first set, the amount of parallelism available in the benchmarks is measured for different numbers of processors. The results demonstrate the amount of parallelism available and how well the parallelism scales for the different benchmarks. In the second set, parallelism is measured while varying the communication type and cost. The results demonstrate the sensitivity of the parallelism to the cost of handling dependences at runtime. In the third set, parallelism is measured while varying the effect of the trace window. The results show the extent to which trace execution needs to be predicted accurately for parallelism to exist. Furthermore, reads are classified and measured based on where they obtain their data to better understand the factors that affect performance.

\textsuperscript{7}These numbers are derived from previous work that characterizes trace behaviour [9]. The graphs in that work indicate that trace execution is in a steady state after $2^{27}$ instructions and that steady state behaviour can be captured in $2^{25}$ instructions.
3.3.1 Effect of the Processor Count

Figure 3.2 shows the abstract speedup of the benchmarks when the default system contains two, four, and eight processors. The benchmarks are listed according to benchmark suite. The first five benchmarks on the left are from Java Grande, the next six are from SPECjvm98, and the last ten are from JOlden. The speedups between benchmarks in the same suite, and therefore of relatively similar type, vary considerably. Thus benchmark performance cannot be easily classified according to type. The values of the different parameters shown in the figure represent a good trade off between overly pessimistic and optimistic assumptions. The effects these parameters have on performance are examined in subsequent sections.

![Figure 3.2: Effect of different numbers of processors.](image)

Six applications, moldyn, raytracer, euler, jess, em3d, and health, exhibit speedups of more than four when the system has eight processors. These applications contain a large amount of trace-level parallelism, which may be exploited on systems with eight or more processors. Seven other applications, search, db, mpegaudio, jack, bh, perimeter, and voronoi exhibit speedups of more than two on both four and eight processors. These applications contain a considerable amount of parallelism. However, the amount of parallelism does not scale to a larger number of processors. The remaining bench-
marks have very little parallelism available. Thus, the results indicate that the majority of benchmarks have enough trace-level parallelism to potentially take advantage of up to four processors, and some benchmarks can potentially take advantage of more than four processors.

An interesting observation is that only two of the JOlden benchmarks exhibit a large amount of parallelism: only *em3d* and *health* have speedups of more than three on the eight processor system. The JOlden benchmarks are designed to have a large amount of parallelism. However, this parallelism is at a very large granularity, much larger than the window size. Therefore, to successfully utilize this parallelism, an approach needs to be developed that can schedule the traces and ensure that dependences are handled over a large number of instructions.

### 3.3.2 Effect of the Communication Model

Figure 3.3 contains the abstract speedup of the benchmarks when the fixed and variable communication costs are varied. Five configurations are shown. The first four have a combination of fixed and variable communication costs of 0 or 20 cycles. The last one has a fixed cost of 80 cycles and a variable cost of 20 cycles.

![Figure 3.3: Effect of the communication cost.](image)

Figure 3.3: Effect of the communication cost.
Three of the Grande and SPECjvm98 benchmarks, raytracer, jess, and jack, show sensitivity to the communication cost. The other benchmarks in these two suites show very little sensitivity. Therefore, the potential parallelism may be achieved even on systems that have considerable communication costs between processors. In contrast, approximately half of the JOlden benchmarks show sensitivity to the communication cost: bh, bisort, health, perimeter, and voronoi. Thus, parallelism may be hindered by systems that have large communication costs.

Figure 3.4 contains the abstract speedup of the benchmarks when the communication type is varied. The effects of trace–trace (TT), trace–instruction (TI), instruction–trace (IT), and instruction–instruction (II) communication are evaluated.

The results indicate that the amount of parallelism is affected significantly by the type of communication used. Only seven benchmarks, moldyn, raytracer, euler, em3d, health, mst, and tsp, show no effect. Thus, being able to communicate dependence information between instructions instead of between the beginnings and ends of traces is important to exploiting the maximum amount of parallelism of the applications.
Furthermore, the results also indicate that the performance of instruction–trace communication is between that of trace–trace and that of instruction–instruction communication, and similar to trace–instruction communication.

Figure 3.3 and Figure 3.4 also indicate that the default configuration, which uses trace–trace communication with a fixed cost of 20 and a variable cost of 0, avoids excessive speedups based on extremely fast communication and presents a good middle ground for the configurations that use trace–trace communication. Thus, although it is not clear which configuration of the abstract system is the most realistic one, the default configuration is a good candidate to be analyzed in more detail.

### 3.3.3 Effect of the Scheduling Window

Figure 3.5 depicts the abstract speedup of the benchmarks for four processors when the trace window size is 128 traces, 1024 traces, and 8192 traces.

![Figure 3.5: Effect of the trace window size.](image)

For most applications, the window size has a small influence on the amount of parallelism available. However, there are several exceptions: mpegaudio, health, perimeter, and voronoi. For these benchmarks, a large number of traces will need to be scheduled together and the control flow between them will need to be predicted accurately. Thus, in
these applications, having a larger window improves parallel performance, but requires better prediction of trace execution. As discussed in Section 3.2.1, such prediction is inaccurate when the window size becomes larger and there are more traces for which execution needs to be predicted. Therefore, an automatic parallelization system may have difficulty reaching the potential level of parallelism available.

Furthermore, although there is no speedup for montecarlo, a careful examination of the benchmark shows that it has parallelism at a very coarse granularity. Enabling the application to run in parallel would require an unreasonably large trace window. Thus, for that benchmark, a more complex analysis would be required to capture the inherent parallelism. Similarly, the JOlden benchmarks are based on algorithms that have very coarse-grain parallelism, and therefore would also require a more complex analysis to capture inherent parallelism. Nonetheless, even without such an analysis, half of the JOlden benchmarks show good speedups.

The speedup that is obtained depends on the specific abstract system used. In particular, the trace window of the system creates boundaries that dictate which instructions can potentially execute in parallel. The window is used to avoid the unrealistic situation of being able to schedule a large portion of the instructions of a program together. This situation can lead to large speedups that cannot be achieved because they are caused by various independent instructions from many parts of the program to be scheduled in parallel. However, using a window also prevents identifying speedups that are due to parallelism at a very large granularity. This limitation can be alleviated by carefully examining the benchmarks to determine whether they contain a large amount of parallelism or by extracting a kernel that can be scheduled effectively when using a trace window.

Given the measured speedups, it is possible to conclude that for most benchmarks, the instruction window can be small, and therefore control flow prediction does not have to be extremely accurate over a large number of traces for parallelism to exist.
3.3.4 Communication Among Traces

In this section, application performance is examined in more detail. Since our processor model is rather abstract, dependence among traces is the main factor that limits parallelism. Therefore, traces and their data accesses are characterized according to the classification in Section 3.2.1. First, a number of distributions is presented: the distribution of the different types of traces, the distribution of reads on these traces, and the distribution of reads on remote traces for the default configuration. Then the same graphs are shown for a system with more processors and a larger window size to show that the patterns hold for multiple abstract systems.

Figure 3.6 presents the distribution of isolated, local, and remote traces. The y-axis contains the percentage of the number of traces in each group relative to all traces in each benchmark. Benchmarks with few local traces tend to perform well. In particular, out of the six benchmarks identified in Section 3.3.1 as having a large amount of parallel execution, five benchmarks, euler, moldyn, raytracer, em3d, and health, have the fewest local traces, 20% or less, relative to all traces. Also, these six benchmarks identified in Section 3.3.1 fall into two categories. One of the benchmarks, euler has a large number of isolated traces without dependences on other traces and therefore the scheduler can easily schedule the traces. The other five benchmarks have many remote traces and therefore have many dependences. However, these dependences were satisfied when the traces were scheduled. Therefore, they exhibit good speedups. Conversely, benchmarks that have a considerable portion of local traces (e.g. mst and montecarlo) all perform poorly.

Figure 3.7 shows the distribution of on-trace, on-processor, and off-processor reads categorized further by which type of trace these reads are on: isolated, local, or remote. In total there are six different types of reads. The y-axis contains the percentage of each type of read relative to all reads. Furthermore, the number above each bar represents the
average number of all reads per trace for that benchmark (these numbers are rounded to the nearest integer in all graphs). Again, these results are for the default system.

In Figure 3.7, most benchmarks that do not exhibit good speedup also have more than 70% of all reads on local traces. This further supports the earlier observation that benchmarks that perform poorly do so because of lack of parallelism (i.e. a considerable number of local traces and reads) and not because of communication overhead. This figure also explains why some benchmarks tend to not be influenced by communication overhead.
type, which is discussed in Section 3.3.2. Benchmarks are not influenced much by communication type if they have a small percentage of reads on local traces (less than 20%). These benchmarks are moldyn, raytracer, euler, em3d, and health. This behaviour implies that most reads are on isolated or remote traces, which are either independent or are dependent in a way that allows traces to execute in parallel in many different combinations. Therefore, the performance of these benchmarks is not affected by the type of communication, and the amount of parallelism exceeds the number of available processors. In contrast, benchmarks that have a large ratio of local reads lack sufficient parallelism. Thus, changes in the type of communication may increase the amount of parallelism, which can in turn be exploited by available processors. Thus, these benchmarks will tend to show sensitivity to the type of communication use.

Figure 3.8 contains the distribution of the three different types of reads on remote traces. The figure further subdivides off-processor reads into critical and non-critical reads. The y-axis contains the four different types of reads and the number above each bar is the average number of reads per remote trace. Applications that have more critical off-processor reads than non-critical off-processor reads also have poor performance. Five benchmarks have this characteristic: compress, bisort, mst, power, and tsp. However, the performance of most of these benchmarks is also hindered by having too many local reads, indicating a lack of parallelism. Since these benchmarks have much fewer critical reads relative to local reads, lack of parallelism has a larger influence on performance.

The seven benchmarks that are hindered by too many critical reads or too many local reads have low speedups in Figure 3.2. Furthermore, all benchmarks except for javac that are not hindered by these factors have good speedups. Therefore, whether or not an application has more than 70% local reads, shown in Figure 3.7, and more critical than non-critical reads, shown in Figure 3.7, is a good indication of whether speedup is present.
The same statistics when the system being simulated has 8 processors, a fixed communication cost of 20 cycles, and a window of 8192 traces are in Figures 3.9 to 3.11. The distributions of traces and reads are similar. This behaviour suggests that the underlying factors that affect performance are relatively independent of the number of processors and window size.

The underlying factors that affect performance are the lack of parallelism and communication overhead. Applications that have an abundance of parallelism (that is a low ratio of local traces/reads) tend to perform well even when critical reads are present, indicating that communication overhead can be hidden. Furthermore, for the minority of applications that do not have abundant parallelism in them, the number of local reads is significantly more than the number of critical reads. Thus, the impact of the local reads is more significant than the impact of critical reads on performance.

### 3.4 Summary

The experimental evaluation of 21 Java benchmarks indicates that most of the benchmarks have enough trace-level parallelism to be run effectively on systems with up to
Chapter 3. The Potential of Trace-Level Parallelism

Figure 3.9: Distribution of traces.

Figure 3.10: Distribution of reads.
four processors. Furthermore, the trace window size has a small effect on the parallelism of most of the benchmarks. However, the amount of parallelism is affected significantly by the type of communication used. Therefore, effective communication is essential to exploiting the full potential of trace-level parallelism. Observations made based on dependence characterization indicate that the main reason for reduced performance is a lack of parallelism. In contrast, communication cost is not a bottleneck because there are only a small number of critical reads. Overall, we conclude that enough trace-level parallelism exists for trace-based parallelization to be a potentially viable approach to improve the performance of Java programs.

Our model of the abstract parallel system is relatively simplistic. Although it does not provide realistic performance numbers, we believe that it provides a reasonably accurate indication of the amount of parallelism that exists among traces. Therefore, we believe that making the simulator more realistic will not change the qualitative conclusions.
Chapter 4

Parallel Trace Execution Model

A sequential program that contains many traces can be executed in parallel by making the traces execute on different execution contexts. In this chapter, we describe an execution model that allows such parallel execution to occur. We first present the model and then show examples that demonstrate how traces can execute in parallel and how a program is transformed to achieve this parallel execution.

4.1 Execution Model

The following is our model for the parallel execution of a program based on traces. A sequential program, which has a single execution context, begins to execute. Initially, no traces are identified for the program. A trace collection system (TCS) monitors this execution and uses the resulting information to record traces. These traces are used to transform parts of the program, which can then execute in parallel. Thus, there are four parts to execution: sequential execution without trace recording, sequential execution with trace recording, transformation, and parallel execution. The model can encompass both an offline and an online feedback directed system.

If the system is an offline feedback directed system, then the execution of the program with a TCS enabled is the preliminary execution. This execution consists of sequential
execution with and without trace recording. At the end of this execution all the traces in the program have been recorded. This complete information can then be used to transform the sequential program into a parallel program. Although traditionally, the transformation occurs before the primary execution, just-in-time compilers, such as those in JVMs, can perform the transformation as part of the primary execution. In the primary execution, the program can execute on more than one execution context. The program’s execution is either sequential without trace recording or parallel.

If the system is an online feedback directed system, then a TCS is enabled throughout execution. Initially the program executes sequentially with no traces being recorded. When parts of the program are frequently executed, traces are recorded. Then the portions of the program that contain traces are transformed to be parallel. After a portion of a program is transformed, it can be executed in parallel. Thus, execution goes repeatedly among sequential execution without trace recording, sequential execution with trace recording, transformation, and parallel execution. Transformation has to be performed throughout the program without knowing all the traces of the program. The appropriate point in execution at which each part of a program should be transformed is difficult to know, and can lead to a lack of parallelism due to waiting too long or not having enough traces. Thus, choosing a good algorithm is important.

In this thesis, our implementation and evaluation is based on offline feedback directed systems. This decision has several implications. The most important implication is that trace collection is not part of the primary execution. Therefore, trace collection overhead does not affect the performance of the parallelized programs. Another implication is that a good algorithm to decide when to transform part of a program is not necessary, since all information is available from the beginning of primary execution. Using an online feedback directed system would require additional research to identify a good algorithm that decides when to transform the program. Furthermore, using an online system would
cause interactions between this algorithm and the transformation. We have decided to use an offline system to avoid these interactions and to focus our work.

Regardless of whether an offline or online system is used, at some point certain parts of a program are executed in parallel. In our execution model, these parallel portions correspond to traces that are recorded and transformed for parallel execution. We refer to this transformation as parallelization. When execution is sequential and reaches one of these traces, this trace's most likely successor trace is identified. This successor trace's own most likely successor is in turn identified. The process repeats until a large enough number of traces are identified. These traces are then distributed by a scheduler to the multiple execution contexts, which execute the traces\(^1\). The scheduler keeps track of the traces and is responsible for distributing them. After the distributed traces execute, if there are more traces to execute then the scheduler will distribute them to the execution contexts. Thus, only already recorded traces can be executed in parallel. Otherwise, execution continues on a single execution context. During sequential execution, when execution reaches another saved trace, execution can become parallel again. The process repeats until the program terminates\(^2\).

To increase the granularity of parallel execution, multiple traces can be grouped together into a single *task*, which we define as a collection of traces that are convenient to group together and form a coarse-grain unit of computation. The traces on a task execute on a single execution context. A task can contain multiple traces with arbitrary control flow between them. In contrast, the control flow of a trace consists of going through instructions one after the other in sequence. Therefore, tasks have more complex control flow than traces. Grouping can be performed when portions of the program are being

\(^{1}\)The instructions on a trace are executed by a single execution context. The execution context may execute some of these instructions in parallel, thus exploiting instruction level parallelism. However, our execution model is not concerned with the manner in which an individual execution context operates.

\(^{2}\)Since the program exits only once, termination will not be on recorded and parallelized traces.
parallelized to allow groups of traces to execute in parallel. There are three benefits to grouping traces into tasks.

First, the overhead of the transitions between traces is reduced. Without tasks, every single trace would have to be put into its own unit of execution, to allow each trace to execute on any execution context. The ability to execute on any execution context requires additional communication overhead due to interactions with the scheduler and having to pass information that is used and defined by the instructions in the trace. The frequent communication of information and transitions between traces would therefore slow execution. When traces are grouped into tasks, the communication overhead between traces on the same task is much smaller because the information between traces can be passed by local variables and there is no interaction with the scheduler when control flow goes from one trace to another trace. Thus, the more traces per task, the smaller the slowdown of execution.

Second, only the order in which tasks execute has to be defined precisely. Traces have control dependences on other traces. In the absence of tasks, the order in which traces execute would have to be defined precisely. However, when grouped into tasks, only the control dependences among tasks need to be resolved ahead of time, and for traces within tasks, the control dependences can be resolved while the traces execute. One approach to minimizing the impact of control dependences is to lengthen traces while trying to keep the increase in early exits small [82]. We avoid the problem of early exits from traces by having multiple traces in a single task. When an early exit occurs from a trace, the probability is high that the control flow will go to the start of a trace. When this trace is on the same task, then there is no exit from the task, and parallel execution can continue.

Furthermore, tasks can be chosen to make the control flow between them easier to predict. The benefit is that hardware support for complex forms of speculation, such as thread level speculation or transactional memory that supports nested transactions across execution contexts, is not needed.
There are many issues that need to be considered when grouping traces together. These issues include making the tasks as large as possible and having many tasks. Having large tasks leads to coarse-grain parallelism and the possibility of effectively amortizing overhead. Having many tasks allows for more parallelism in two ways. First, there are more tasks to schedule. Therefore, the tasks can be scheduled on more execution contexts and the scheduler can choose between a larger combination of tasks. Second, there can be fewer dependences between smaller tasks. For example, consider tasks that are in a loop. There may be dependences between different iterations of the loop that limit the amount of available parallelism. If the tasks are small enough for there to be multiple tasks between the dependent iterations, then these tasks can execute in parallel. However, if the tasks are bigger and there are dependences between them, then the dependence forces the tasks to be sequential\textsuperscript{3}. The approach that we have taken, which tries to balance the issues, is described in Section 5.1.

Third, scheduling overhead is reduced. Tasks are larger in size and fewer in quantity than traces. Thus, scheduling can take less time and can be performed less frequently.

\subsection{Parallel Trace Execution Example}

An example that illustrates traces executing in parallel under our execution model is shown in Figure 4.1. The left side of the figure contains part of the initial CFG that contains traces. In this example, we assume that trace recording for at least part of the program has already been performed by a trace collection system, as described in Section 2.5.1, and that traces $T_1$ through $T_6$, which have no dependences between them, are recorded. The traces are indicated by large arrows. Further, we assume that the

\footnote{Consider a loop with four iterations and a dependence between the first and last iteration. Further, assume that no forwarding of the dependent data can be done between the middle of tasks. If there are four tasks, one per iteration, then the first three tasks could execute in parallel. If there are two tasks, one per two iterations, then the two tasks would have to execute sequentially. Our implementation does not take such dependences into account (see Section 7.5), and addressing this shortcoming is left for future work.}
traces are grouped into three tasks that contain $T_1$, $T_3$ and $T_4$, and $T_6$ respectively; that the most likely execution flow is $T_1$, $T_3$, $T_4$, and $T_6$; and that the actual execution follows the most likely execution flow\textsuperscript{4}. The most likely path of execution is indicated by vertical lines on the left sides of the basic blocks. Only the basic blocks and edges that are part of traces are shown. The other parts of the CFG and edges to basic blocks not on traces are not shown. The dots indicate parts of the CFG before and after the traces.

![Figure 4.1: Traces executing according to the proposed execution model.](image)

The right side of the figure contains part of the execution of the program on a system with two execution contexts. Execution is initially sequential and not on traces. This execution is indicated by dots. Once execution reaches a task, this task and its identified successor tasks are executed in parallel. When an execution context is idle, it invokes the scheduler, which finds appropriate tasks for the context to execute. The scheduling and starting of tasks is indicated by an oval. The scheduler assigns tasks to the two execution contexts. The sequential program order of the traces is kept track of, and thus

\textsuperscript{4}This example is only for illustration, and the specific grouping of traces is arbitrary.
what to execute next is always known. The task execution needs to be performed in a way that ensures that dependences are handled properly, and that no dependences are violated. Since there are no dependences between the traces in the example, the tasks can be executed in the order shown in the figure. Once the task on the first execution context completes, another task is scheduled on that execution context. Once the four traces finish executing, the subsequent instructions in sequential program order are not on traces. Thus, sequential execution resumes.

\section*{4.3 Transformation Example}

The example in the preceding section assumes that a portion of the program has been parallelized. This section shows an example of how the parallelization occurs. The parallelization turns the CFG of a sequential program that executes on a single execution context into a CFG for multiple execution contexts. This new CFG has to have instructions that ensure that tasks are scheduled properly and that all dependences are handled.

The additional instructions that ensure that tasks are scheduled properly can be grouped into three categories:

- instructions that start and end tasks, which package the tasks to enable their execution by any execution context chosen by the scheduler;

- instructions that start and end execution on each execution context; and

- instructions that schedule the tasks on multiple execution contexts.

The CFG that is a result of adding these three groups of instructions allows for programs to follow the previously described execution model.

In this example, dependence handling is not an issue, and the instructions that handle dependences are not shown. These instructions are described later in Section 5.3.
The parallelization transformation is demonstrated in Figure 4.2, which contains the initial sequential CFG of a program and the parallel transformed CFG of the same program. The program is the same one used in Figure 4.1, and the left side of the figure contains the same sequential CFG.

Figure 4.2: Control flow graph transformation.

The right side contains the transformed CFG, when the traces that are likely to execute have been grouped into tasks and the tasks have been packaged into units of parallel execution. Oval nodes are aggregate nodes that represent complex behaviour. The oval nodes represent the scheduler and the beginning and end of each task. The exact details of the behaviour are not shown, and will be described in later chapters. Two execution contexts exist and the CFG has corresponding start and end nodes.

The first context can execute both sequential and parallel parts of the program, while the second context can execute only parallel parts of the program. The beginning of
the first context contains instructions for the sequential execution of the program and is indicated by a dotted line. At the point execution reaches the beginning of trace $T_1$, the program can begin executing in parallel. Thus, instructions that schedule tasks are called, which is indicated with an edge to the scheduler oval node. The scheduler is responsible for determining which tasks to execute in the execution context. Eventually, after the scheduler has executed all appropriate edges, sequential execution will resume, which is shown by the edge from the scheduler to the lower dotted line.

The second context immediately starts executing scheduler instructions, which will wait until a task can execute in the second context. The scheduler will wait until there are tasks to execute. When the program terminates the scheduler instructions will go to the end of the second execution context.

The scheduler, which executes on all the execution contexts is responsible for determining which tasks to execute in each execution context and when to stop executing in parallel. The scheduler will ensure that the appropriate tasks are executed at the appropriate point in execution. The scheduler may be called from inside of tasks as well, although that is not the case in this example.

This example shows a portion of the static instructions of the program, while the previous example in Section 4.2 shows a portion of the dynamic instructions that correspond to one possible execution of these static instructions. In particular, the dynamic instructions show the scheduler deciding that the first execution context should execute $Task_1$ and $Task_3$ while the second execution context should execute $Task_2$. The resulting execution consists of executing traces in parallel, thus the goal of being able to parallelize a sequential program is achieved.

The above description of our execution model shows that a number of steps need to be performed to use traces to allow a sequential program to execute in parallel. The steps consist of grouping traces into tasks, packaging tasks into parallel units of execution, inserting instructions that handle dependences and ensure that they are not violated,
and scheduling these packaged tasks when they execute. These four steps correspond to
the challenges described in the introduction, and are addressed by our framework.
Chapter 5

Trace-Based Parallelization Framework

We have created a trace-based parallelization framework that allows sequential programs to be executed in parallel. The framework consists of four components that perform the four activities listed at the end of the previous chapter: grouping traces into tasks, packaging tasks, inserting instructions that handle dependences, and scheduling these packaged tasks. We refer to these components as the grouping component, packaging component, dependence component, and scheduler respectively.

5.1 Grouping

The first step of parallelization is to use traces to create tasks that will execute in parallel. The creation of tasks has to consider two competing objectives: amortize the parallel overhead by having coarser grained tasks and allow a large amount of parallelism by having finer grained tasks. Furthermore, the control flow between tasks should be as predictable as possible. The following elaborates on these objectives.

Parallel overhead can be amortized by making tasks have a large number of instructions and thus increasing task granularity. Assuming that the overhead of executing one
task is fixed, the more instructions are executed in a single task, the lower the overhead is as a percentage of task execution. However, since the number of instructions executed in a single execution of a program is fixed, executing more instructions per task leads to fewer tasks being executed, and may reduce parallelism.

Allowing a large amount of parallelism is important because the more parallelism exists, the higher the potential speedup. Parallelism can be increased by increasing the number of tasks that are executed. Therefore, a balance needs to be found between the amount of parallelism and the amount of overhead amortization.

Having predictable control flow between tasks is important, because that simplifies the parallel execution of tasks. If the control flow is known completely ahead of time\(^1\), then the tasks can just be executed in parallel, without having to worry about incorrect execution of a task. Predictability is a relatively orthogonal issue to group size. Instead, predictability is determined by the predictability of control flow at task boundaries. Since cycles in a CFG indicate that the same code will execute multiple times, they are an indicator of predictable performance. Therefore, task boundaries are made along cycles. That is, the tasks are made to begin at the starts of cycles and finish at the ends of cycles\(^2\). For iteration, a cycle is formed when control flow goes from the end of the loop to the beginning. For recursion, a cycle is formed when control flow goes from a call of a recursive method back to the beginning of the method. Therefore, these cycles encompass both iteration and recursion.

Two benefits arise from making task boundaries along cycles. First, control flow not on cycles is kept inside of tasks. Control flow that is not part of cycles will tend to consist of decisions in the program logic. Although these decisions may be highly correlated over

---

\(^1\)If the control flow is not known ahead of time, then it must be handled properly. This issue is discussed in Section 5.3.

\(^2\)Only parts of a program that contain cycles can be parallelized using such an approach. Although potentially a limitation, in practice all programs that have non-trivial execution times must have cycles. A program without cycles would have each static instruction executing only once. This leads to extremely large programs, which are not practical.
small periods of time, if the behaviour of the program changes, then the decisions will change. Furthermore, some decisions will be hard to predict, for example if they are based on a randomly generated variable. When this control flow is inside a task, it does not affect the predictability of control flow between tasks. Second, control flow between tasks that occurs on cycles tends to follow a predictable pattern. Control flow at cycles tends to consist of going along the cycle a large number of times, until at some point the cycle is not followed. This pattern occurs both for iteration and recursion. For iteration, a loop will execute a number of times until it exits. For recursion, a method will be called multiple times until recursion ends. Prediction can be aided by analyzing the program and using runtime information to find out how often the cycles will be followed. For example, if a cycle is due to a loop, analysis may identify the induction variable of the loop and the values of the variable. This identified information can be used at runtime to accurately predict how many times the loop will execute.

We use a graph-theoretic approach to perform the analysis to identify tasks. To perform this analysis, we use a graph of traces, which is a graph that contains all traces and control flow edges between them. Each vertex is a trace, and each edge corresponds to a control flow path from instructions on one trace to the beginning of another or possibly the same trace. The graph of traces is the input to the grouping component, and tasks are generated based on analyzing this graph.

The grouping component identifies the strongly connected components (SCCs) of this graph of traces and then uses the SCCs to create tasks. An SCC is a graph such that all pairs of nodes have paths to each other. The vertices of each SCC are traces, and the edges correspond to control flow paths from instructions on one trace to another trace. Each SCC is a subgraph of the graph of traces. SCCs are used to identify large subgraphs of the graph of traces that contain cycles, and can be further subdivided to create tasks.

\[^3\]Although a single trace with no trace exit pointing to a trace start, including the trace’s own start, is a trivial SCC, we do not consider it to form tasks because there is no cycle and such a trace, with sequential code executed before and after it, can only be executed sequentially.
Using SCCs is beneficial because SCCs divide the graph of traces into disjoint parts without cycles. Any task that is formed should be part of a single SCC instead of spanning multiple SCCs. The reason is that there are no cycles between SCCs, which has two consequences.

First, a task that spans two SCCs cannot have its beginning and end on a cycle. For example, consider a task that contains two consecutive traces, \( T_1 \) and \( T_2 \), that are on two separate SCCs, \( SCC_1 \) and \( SCC_2 \), respectively. This task will start in \( SCC_1 \) and end in \( SCC_2 \). Thus, this task will not have an edge going back from the task’s end to the task’s beginning.

Second, a task that spans SCCs will prevent the traces that it contains from being part of other tasks. For example, consider a second task being created that contains several traces in \( SCC_1 \). If this second task needs to contain \( T_1 \) to be along a cycle, this second task will not be able to get that trace because the first task already contains it. Thus, this second task will not be along a cycle as a result of the first task containing traces from multiple SCCs, including \( SCC_1 \).

Therefore, any task formation algorithm that creates tasks along cycles will need to take into account SCCs, even if it does not compute SCCs. The following example shows that not taking into account SCCs can lead to groups that cannot be used to execute traces in parallel. Consider the graph shown in Figure 5.1. The graph contains four traces in code order, Traces 1 through 4. There are two cycles. The first cycle starts at Trace 1 and ends at Trace 3. This cycle has a backward edge in code order from Trace 3 to Trace 1. The second cycle starts at Trace 2 and ends at Trace 4. This cycle has a backward edge in code order from Trace 4 to Trace 2. If groups of traces are formed along the backward edges of cycles by taking all traces starting at the target of a backward edge to the end of this edge, then there are two possible groups. When the first cycle is initially considered, Trace 1, Trace 2, and Trace 3 are in one group, while Trace 4 is in a second group. Alternatively, when the second cycle is initially considered, Trace 2, Trace...
3, and Trace 4 are in one group, while Trace 1 is in a second group. In both cases, one of the groups contains a single trace by itself, which cannot be executed multiple times. Thus, this group of a single trace cannot be used to execute traces in parallel.

This problem can be avoided by starting with SCCs. The graph has two SCCs. The first SCC contains Trace 1 and Trace 3. The second SCC contains Trace 2 and Trace 4. Using the same algorithm within the SCCs would results in the formation of two groups. The first group is created when the algorithm is used on the first SCC. This group consists of Trace 1 and Trace 3. Similarly, the second group is created when the algorithm is used on the second SCC. This group consists of Trace 2 and Trace 4. Both groups contain multiple traces that can execute multiple times together. Thus, both groups may be used to execute traces in parallel.

![Figure 5.1: Example of part of a CFG with four traces on two SCCs.](image)

Furthermore, a task formation algorithm that works on individual SCCs instead of the entire graph will have a smaller graph as input. Assuming that the algorithm has a higher complexity than SCC identification, grouping will be faster if the graph is divided into SCCs first. Performance is important if the algorithm is performed at runtime in the primary execution.
To calculate SCCs, the graph of traces needs to be generated and then SCCs can be identified based on this graph. The algorithm to identify the edges on the graph of traces depends on the underlying implementation and is described in Section 7.2. However, the general idea is to go through all the control flow instructions in each trace, and if the target of the instruction is the first instruction of another trace, then there should be an edge in the graph from the first trace to the second trace. The starting point of each trace is the method and bytecode index of the first instruction of the trace. Furthermore, each edge is associated with the type of the corresponding control flow instruction, and the traces are ordered based on the starting points. Once the edges between traces are identified, we use Tarjan’s algorithm to identify the SCCs [30].

5.1.1 Task Identification

Once SCCs are calculated, tasks need to be identified. The simplest approach is to have each SCC be a task. However, that only allows parallelism among SCCs, which is unlikely and will miss parallelism within an SCC. The second simplest approach is to have all traces in the SCC in one task, and then each task would be a single execution from the top to the bottom of the SCC\textsuperscript{4}. This approach works well when the SCC is a loop nest. However, the approach does not work well for recursion, because recursive programs create work over time, and the amount of work is not known at the beginning of the SCC.

Therefore, we take a different approach to task formation, and partition each SCC into tasks that are both coarse-grained and have control flow along cycles. This is accomplished by categorizing edges in each SCC into

\textsuperscript{4}The top is the earliest part of the SCC in code order, which corresponds to the order in which instructions appear in the method (i.e. the bytecode index for Java), and the bottom is the last part of the SCC in code order.
• forward edges that represent forward control flow (including all control flow from returns),
• backward edges that represent backward control flow, and
• implicit edges that represent a connection between calls and the instructions after the call in code order, which will be executed after the method being called returns.

Backward edges are differentiated from forward edges for two reasons. First, backward edges indicate that execution is about to go to an earlier part of an SCC. Therefore, these edges indicate which code is executed multiple times, and thus that more parallelism can exist. Second, the source of the backward edge is the last code executed before going to an earlier part of an SCC, and the target of the backward edge corresponds to the start of that earlier part. Thus, these edges can be used to identify the largest possible sequences of code from earliest to latest on the SCC. Therefore, a good approach to deciding where tasks should start and end within an SCC is to make tasks start at targets of backward edges and end at sources of backward edges.

However, the body of a task should not end if the same instructions always execute after that task. Instead, the task should continue, and these instructions should be made part of this task. The benefit of extending the task is that more instructions will execute inside the task without having to cross task boundaries. Therefore, fewer transitions between tasks occur, and overhead is reduced. This scenario occurs when the source of a backward edge is a call instruction and the method that is called returns, the instruction after the call in code order will always execute. Therefore, implicit edges are used to identify locations at which tasks can be extended. The implicit edges go from calls to the subsequent instructions in code order, and tasks should not end at the sources of backward cycles if an indirect edge can extend the task.

All edges that are not backward or implicit edges are categorized as forward edges. Forward edges are those edges that are not used by the algorithm to determine tasks. We categorize returns as forward edges because the targets of implicit edges are also targets
of returns. Since the targets of implicit edges should not start tasks, returns are not treated as backward edges, which start tasks$^5$.  

Formally, given an SCC that contains a set of vertices called traces and a set of edges called edges, three sets of edges can be computed that contain forward, backward, and implicit edges. The backward edges are those that are backward in code order and are not returns. The forward edges are edges that are forward in code order or are returns. The implicit edges are edges from the sources of backward edges that correspond to invokes to the traces that appear in code order immediately after the invokes. Thus:

\[
\text{backward-edges} = \{e | e \in \text{edges}: e.target \leq e.source \text{ and return}(e) = \bot \}
\]

\[
\text{forward-edges} = \{e | e \in \text{edges}: e.target > e.source \text{ or return}(e) = \top \}
\]

\[
\text{implicit-edges} = \{(e.source, t) | e \in \text{backward-edges} \text{ and implicit-target}(e) = t \}
\]

When the two traces have starting points with the same method and the bytecode index of the first is smaller or equal to that of the second, $\leq$ is true ($\top$) and $>$ is false ($\bot$). Otherwise, if the two traces have starting points with different methods or the bytecode index of the first is greater than that of the second, $\leq$ is false (and $>$ is true). The boolean return function indicates whether an edge is based on a return statement or not. The implicit-target function takes an edge as input. When that edge corresponds to an invoke, the function returns the trace that starts right after that invoke (and false otherwise). If the edge does not correspond to an invoke or there is no trace then $\bot$ is returned.

The above notion of a backward edge is in contrast to more traditional definitions of backward edges based on depth first traversal [57]. The reason for us adopting this non-traditional definition is threefold. First, the code order is used instead to avoid problems with loops that are optimized to have a single control flow instruction at the end. The traditional definition would indicate that this last instruction is not the backward edge,

$^5$Although the returns could be put in their own edge category, that category would be redundant. The reason is that it would just represent edges that are not used by the algorithm that determines tasks, which is the same as forward edges.
which is inconvenient for our transformations. These transformations are simpler if the backward edge is at the end of all the instructions in the loop. For example, for a loop that after transformation contains two instructions, \( L0: i++ \) and \( L1: \text{if} \ (i<5) \ \text{goto} \ L0 \), with the loop starting at the second instruction (e.g., in Figure 2.2 on page 21), the traditional definition would require a back edge starting at the \( i++ \) instruction. Therefore, a task would be made to end at the \( i++ \) instruction and a new task would begin at the \textit{if} instruction. The result would be a dependence between the two tasks, which can be removed by adjusting the task boundaries. With the backward edge starting from the \textit{if} instruction, the task boundaries can stay as is, although a copy of the \textit{if} instruction needs to be created to ensure no tasks execute when \( i \) is too large. Thus, transformation is simpler\(^6\).

Second, using the traditional definition would lead to higher trace collection overheads. The reason is that a control flow graph would need to be constructed if it does not exist and a depth-first traversal would need to be performed. However, just using code order does not require these additional steps, and thus, can lead to improved trace collection performance\(^7\).

Third, since traces can span multiple methods, our analysis is interprocedural, and the traditional definition of backward edges leads to hard-to-deal-with edges. In particular, calls could be identified as backward edges. Consider an example shown in Figure 5.2. Two methods, \( a() \) and \( c() \) call method \( b() \). If a depth-first traversal starts with method

---

\(^6\)This loop optimization is performed by some bytecode compilers. The bytecode compilers seem to not make other optimizations that transform branches. Therefore, we have found that branches not related to loops always jump forward. Our framework transforms the program based on the bytecode indices before any optimizations are performed by the optimizing compiler. Therefore, the order used is independent of what the optimizing compiler does. We have found no issues with using code order. However, if the effect of optimizations needs to be considered, then the algorithms and definitions will need to be modified accordingly.

\(^7\)In our implementation, the instructions to communicate with the trace collection system are inserted by a non-optimizing compiler that generates code for each instruction separately. The compiler does not use a control flow graph, does no optimizations, and does not perform any analysis. Using a non-optimizing compiler is consistent with the way that an online trace collection system would work. Having the compiler perform the additional steps would be a significant change with negative effects on performance.
c(), then goes to method b() via the call, and finally goes to method a() via the return, then the backward edge of the loop in method a() will be the invoke of method b(). Having a call as a backward branch of a loop would be difficult to handle. Therefore, we avoid such a scenario by using a different definition for backward edges. This is similar to choosing backward edges from a set of retreating edges in an irreducible graph based on these edges having sources and destinations being in the same method.

![Diagram of control flow between three methods](image)

**Figure 5.2:** Example of control flow between three methods.

The backward, forward, and implicit edge sets can be used to identify the start and end points of tasks, as well as the locations at which the tasks are forked:

- \( \text{task-starts} = \{ t | \exists e \in \text{backward-edges}: e.\text{target} = t \} \)
- \( \text{task-ends} = \{ \text{last-trace}(\text{traces}) \} \cup \{ e.\text{source} | e \in \text{backward-edges} \land \forall f \in \text{implicit-edges}: f.\text{source} \neq e.\text{source} \} \)
- \( \text{task-forks} = \{ e | e \in \text{edges}: e.\text{target} \in \text{task-starts} \} \)

The last-trace function returns the last trace of a set. The task starts are traces that are targets of backward edges. The starts of the tasks correspond to the starts of the
traces. The task ends are the last trace on the SCC and the sources of backward edges that are not sources of implicit edges. The ends of the tasks correspond to the ends of the traces. The task forks are edges that point to the task starts. A task contains all the traces from the start trace to the end trace that are not part of a different task, which can be identified by checking the task-starts and task-ends sets when going through the traces.

The algorithm that computes these six sets given the traces and edges of an SCC is shown in Algorithm 1.

**Algorithm 1** Algorithm to find tasks from SCCs.

```plaintext
process-scc(traces, edges)
    forward-edges = {}
    backward-edges = {}
    implicit-edges = {}
    for each e in edges
        if e is forward then add to forward-edges
        else add to backward-edges
        for each e in backward-edges
            for each t in traces
                if t is implicit target of e.source then
                    add (e.source, t) to implicit-edges
    task-starts = {}
    task-ends = {last trace in traces}
    task-forks = {}
    for each e in backward-edges
        add e.target to task-starts
        if e.source not a source of an edge in implicit-edges then
            add e.source to task-ends
    for each e in edges
        if e.target in task-starts
            add e to task-forks
```

The algorithm only uses the types of edges, and does not use any profile information about the frequency of the edges. We have made the decision to only use the types of edges for two reasons. The first reason is that the additional information added is not
expected to contribute in a significant way. All traces represent frequently executed sequences of instructions, and therefore are important for task selection. Although we have no evidence, we expect that knowing that a particular edge between traces is more frequently executed than another would yield little benefit when all the traces are frequently executed. Providing such evidence would require an elaborate study that is beyond the scope of this work. The second reason is that using profile information would add extra complexity. The complexity would result in an algorithm that is longer, more complex to reason about, and possibly slower to perform. Furthermore, since additional profile information would need to be collected by the trace collection system, the possibility of having a low overhead online trace collection system would be reduced. Therefore, we prefer to use a simple task identification algorithm.

5.1.2 Iterative Example

The following example shows how tasks can be formed in an iterative program. Figure 5.3 contains a CFG on the left and an SCC after edge categorization on the right. The SCC has a single trace and one backward edge, indicated with a solid line. One task is formed, and consists of the one trace.

![Figure 5.3: Categorizing edges of an iterative program.](image)

Figure 5.4 contains the CFG of the task with *regular edges* and *fork edges*. The regular edges correspond to the control flow, and have sources and targets within the
same dynamic task. All edges between dynamic tasks need to be removed, since edges that go between tasks should not exist; instead, there should be edges going to and from the scheduler to invoke and return from the scheduler at task boundaries. However, the control flow is easier to follow if the scheduler is not shown, and edges are shown between tasks. Thus, we add fork edges that indicate that a new task will be spawned and subsequently executed.

![Figure 5.4: CFG with a task in an iterative program.](image)

Additional fork and join instructions are added to the CFG. The forks are added where control goes to the beginning of the task. Therefore, a fork is added after the `i=0` instruction and after the `if` statement. The fork is in a different basic block to indicate that it is executed only when the condition in the `if` statement evaluates to true. The forks have corresponding joins, which are used to enforce dependences by waiting for the forked execution to complete. Join insertion is discussed later, in Section 5.2, although intuitively, joins are added before variables with data from forked tasks are used and before control flow leaves the method. The join for the first fork, labelled L0, occurs before the end of the method. Since a dependence between tasks exists on variable i, the join for the second fork, labelled L1, occurs before i is used\(^8\). For the first task,

\(^8\)This join exists only for illustrative purposes. Our framework would identify i as an induction variable and would remove the dependence.
the join will be ignored since no previous forks at L1 were executed. At the end of a
task, execution will go back to the scheduler. Although not shown, the task needs to be
packaged in a way that allows the variable i to be passed between executions of the task,
which will be addressed in Section 5.2.

5.1.3 Recursive Example

The following shows how traces that are part of a recursive program are grouped. The
left side of Figure 5.5 contains an interprocedural CFG with four basic blocks that also
represent four traces. This graph also corresponds to an SCC. The calls to $f()$ end their
basic blocks because control flow goes to the beginning of method $f()$. The right side
contains the CFG with the backward, forward, and implicit edges categorized. These are
indicated with solid, dotted, and dashed lines respectively. Based on the edges, a task
can be formed. The task start is at the top of the CFG. All sources of backward edges
are also sources of implicit edges. Therefore, they do not indicate task ends. Instead the
task end occurs at the end of the SCC. Therefore, there is one task that contains all four
basic blocks/traces.

Figure 5.6 shows the CFG of the task. Both regular and fork edges are shown. The
method calls are replaced with forks, and joins are inserted just before the return. In
the original CFG the target of the calls to method $f()$ is the top basic block. However,
in Figure 5.6, the forks send tasks to execute on other execution contexts, and there are
no control flow edges from them unless they are added. Therefore, the implicit edges
are added as regular edges. Since the target of the implicit edge is whatever instruction
is directly after an invocation, adding implicit edges ensures that execution proceeds
properly. One edge is added from the first fork to the second fork, and one edge is added
from the second fork to the return.
Figure 5.5: Edge categorization of a recursive program.

Figure 5.6: CFG of a recursive task.
5.1.4 Recursive and Iterative Example

The following example shows how tasks are formed both for iteration and recursion. Figure 5.7 contains an interprocedural CFG on the left and an SCC after edge categorization on the right. Each basic block corresponds to a trace. Backward, forward, and implicit edges are indicated with solid, dotted, and dashed lines respectively.

Tasks are then formed based on the edges. Although three backward edges exist, two have the same target, and only two tasks are formed. The first task starts at the beginning of the SCC and the second task starts at the beginning of the loop that the second trace is on. This second task ends at the backward edge, and thus contains one trace. The first task goes past the second task and continues until the SCC’s end. All edges that have the starts of the tasks as targets will need to have forks added.

Figure 5.8 contains the tasks with regular and fork edges. Fork instructions are added to indicate the sources of fork edges, and corresponding joins are inserted. Both forks
and joins are added in the same locations as in the previous examples. Although not shown, the tasks need to be packaged in a way that allows the variable $i$ to be passed to task 2. An additional regular edge needs to be inserted for the first task. The reason is that creating the tasks causes there to be a discontinuity in the control flow of the first task. This discontinuity can be resolved by adding an edge between the appropriate consecutive blocks/traces. The resulting CFG also has fork instructions inserted at the appropriate points. In the figure, a fork 2 instruction has been added to the end of task 2 and two fork 1 instructions are added to task 1.

Figure 5.8: CFG of tasks.
Chapter 5. Trace-Based Parallelization Framework

5.2 Extraction and Packaging

Once tasks and fork points have been identified, the tasks can be extracted from the rest of the code and packaged into units of execution. The tasks must be packaged in a way that allows them to be executed by a JVM. A number of factors need to be taken into consideration. The unit of compilation and execution in Java is a method. Therefore, all tasks need to be packaged as methods. Furthermore, all information in method parameters and local variables needs to be communicated across forks and thus must be packaged in a way that allows the information to be used effectively across forks on different execution contexts.

The packaging component takes as input the tasks and fork points from the grouping component to put tasks in their own units of execution and ensures that all variables are passed properly to and from tasks. Furthermore, output of the dependence component, described in Section 5.3, is used to find out which tasks are associated with induction variables and where joins for each fork should be inserted. A join will wait until the execution resulting from the corresponding fork completes. Thus, joins are useful to enforce dependences between the forked execution and instructions after the join.

To package in a way that considers the above factors, six steps need to be performed for each task that exists in a method: two to extract them, and four to package them. The first is to identify the section of the CFG that needs to be packaged. The second is to ensure that the section is put in its own unit of execution, which in Java is a method. The third is to add prologue and epilogue instructions that pass information to and from the task. The fourth is to ensure that forks, which are identified by the grouping component, are inserted at the appropriate points in the CFG to allow the task to be executed in parallel. As will be discussed in Section 5.3, induction variables may be associated with tasks. Thus, if there are induction variables associated with a fork, then the fork needs to save information, such as increments, lower bounds, and upper bounds, about the variables. Also, the fork is scheduled as early as possible, to ensure maximum
distance between the fork and any potential joins. This code motion is described in Section 7.4.2. The fifth is to add instructions at each fork that pass parameter and local variable information to the task. The sixth is to add instructions at each join that receives information from the task. Joins are needed to enforce dependences, and join insertion is based on dependence information, which is discussed in Section 5.3.

An example is shown in Figure 5.9. On the left side of the figure is a CFG with four basic blocks, $A$ through $D$, shown. The dotted line represents other basic blocks. $D$ contains the first use of anything written in $B$ and $C$. Further, $B$ and $C$ should be put into a task. On the right is the transformed CFG. The two blocks $B$ and $C$ are put in a task, which needs to be a separate unit of execution. A prologue, an epilogue, a fork, and a join are added along with appropriate instructions to pass and retrieve information. Communication that needs to be performed between execution contexts is shown by dotted lines. The execution will start with $A$, then a task will be forked, which would make the task execute on a different execution context. Execution will continue until $D$ requires a join to occur because it uses what is written in $B$ and $C$. At that point, a join is performed with the forked task.

Figure 5.9: Packaging of a task.
The information that is passed to the task consists of values of any variables that may be read by the task and the initial values of variables that may be written by the task. The reason for passing in initial values of variables that may be written is that the task may end without certain writes occurring, and yet a value needs to be written back. In this case, the passed in value is used. The retrieved information consists of the final values of all variables that may be written by the task.

When the start of a task corresponds to the start of a recursive method, two complications arise from using the six-step approach, which would put the task in its own separate unit of execution. The result of the six-step approach for a recursive program is illustrated in Figure 5.10. The left side contains the original CFG, in which method \texttt{g()} calls a recursive method, \texttt{f()}, from the recursive example shown previously in Section 5.1.3. The right side contains the transformed code with the task being in its own unit of execution. The solid edges denote regular control flow while the dotted edges indicate fork edges.

![Figure 5.10: Packaging a recursive task.](image-url)
The first complication is that virtual method invocation allows for methods other than the one expected to execute. For example, there may be a method \( f() \) from a different class. Therefore, task execution must follow the virtual method execution to ensure that the right method is executed. However, if the task is in its own separate unit of compilation, which virtual method to execute is not kept track of.

The second complication is that there are situations when a method that is already optimized and currently executing should have a fork inserted into it. In such a situation, the fork cannot be inserted\(^9\), and the result is that the task would not execute, thus forcing execution to be sequential. For example, it may be not possible to modify \( g() \), and therefore the task would never execute.

If instead, the task is left inside of the method, \( f() \), then the proper virtual method will be followed. Thus, the first complication is addressed. When the method is recompiled and forks are inserted, although the existing execution will not contain the fork, the next invocation of the method will contain the forks. Thus, the second complication is also addressed.

Therefore, for such tasks, the best approach is to not put them in a separate unit of compilation. Thus, the second and third steps of the six step approach are not performed: the task is not put in its own unit of execution, and prologue and epilogue instructions are not added. The other steps, which identify the section of the CFG and deal with forks and joins still need to be performed.

Furthermore, to improve performance, two versions of a task can be created. A slow version that can fork other tasks, and a fast version that does not fork other tasks and thus limits execution to one execution context. The scheduler can decide when it is appropriate to use the different versions, as will be explained in Section 5.4. A test is

\(^9\)Although on-stack replacement may be possible, sometimes almost the entire stack would have to be changed, which would result in a large overhead.
used to control which version is executed\textsuperscript{10}. The result is that execution can still be forked and that execution that is limited to one execution context is fast.

An illustration is shown in Figure 5.11. The CFG is the same one used in the previous example. The method $f()$ now contains a test and two versions of the task inside of it. The slow version has added overhead since it will fork tasks. The fast version executes in a sequential manner without the overhead. By default execution will start in the slow version and at some point, once enough parallelism is realized and the granularity of tasks is reduced, execution will switch over to the fast sequential version. The combination of the two versions ensures correct execution that is as fast as possible.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure511}
\caption{Packaging of two versions of a task.}
\end{figure}

\textsuperscript{10}In our implementation, we have found that always performing the test results in a small overhead. The overhead is smaller than that required to resolve a virtual method. If the overhead needs to be removed, the method to be called is known, and only the fast version is to be used, then a special fast method can be created without the test. We have not implemented the additional logic required, since our experimental evaluation shows that speedups are obtained when the test is always performed.
5.3 Dependences

Once tasks are packaged, the parallelization framework must add instructions to the packaged tasks for handling dependences. Handling dependences is difficult because, in general, dependences cannot be known ahead of time, and therefore cannot be easily handled. Dependences can be handled in one of two ways, either by removal or by enforcement. We deal with dependences by explicitly enforcing dependences due to forks, removing dependences on induction and reduction variables, and using HTM to enforce all other dependences.

The forking of tasks creates dependences that are known ahead of time. Therefore, our framework handles this situation by inserting joins, which wait for the corresponding forks to complete. These joins are inserted at the appropriate points of execution that ensure that dependences are handled. When a forked task has information to pass back to the forking task, then joins are inserted before that information is used\(^\text{11}\). The analysis used to identify where to insert joins is described in Section 7.4.2.

Dependences on induction and reduction variables are removed. Our framework deals with simple inductions and reductions by precomputing the induction values and privatizing and combining the reduction variables. When a join occurs that has a corresponding reduction variable, then the join must combine the reduction variable values. The dependence component identifies the induction and reduction variables on the tasks, and then associates the tasks with these variables. The packaging component can use this information to insert appropriate instructions to pass appropriate information about these variables, which can be used during the execution of the tasks to identify how many tasks to execute and which values of induction variables to use. The type of induction and reduction variables handled by our implementation of the framework are described in Section 7.4.1.

\(^{11}\)The current analysis is intraprocedural. Therefore, returns and calls to other methods need to have joins inserted before them as well.
We use speculation to enforce all other dependences. Execution proceeds as if there were no dependences. However, the dependences that occur at runtime are kept track of. If a dependence is violated then the execution must be undone and performed again. Control dependence violations can be enforced by causing speculative execution to abort. The easiest approach would be to cause a data dependence violation whenever control flow goes in a direction that is not predicted. Thus, the control dependence is turned into a data dependence. The program would also have to be modified to ensure that execution will proceed regardless of what control flow is taken, for example by forcing execution to proceed sequentially. The experimental evaluation did not require us to deal with control dependence violations, and therefore we leave dealing with control dependence violations for future work.

A popular approach to speculation is transactional memory. Although software transactional memory has large overheads, hardware transactional memory (HTM) is showing promise as a potential low overhead approach to speculation. Therefore, our framework relies on HTM to handle all dependences that are not removed or not part of the information passed by forking tasks. The details of how transactional memory can be used with our trace-based framework is described in Chapter 6.

### 5.4 Scheduling and Execution

The role of the scheduler is to decide which tasks to execute on each execution context. The scheduler needs to make this decision for each task that is forked. Furthermore, for a task that is associated with an induction variable, the scheduler will identify the values that the variable takes. Then the scheduler will fork and schedule all tasks associated with these values. The benefit of forking and scheduling the tasks together is the reduction of

---

12Control dependences are not an issue because the task formation algorithm generates tasks whose execution can always be predicted with induction variables or recursive method calls. If traces are collected online or different benchmarks are used, then control dependence violations may occur.
overhead. The reason for the scheduler dealing with induction variables is that sufficient information exists to identify the values of the induction variable only when the task is about to execute, which corresponds to when the task is scheduled.

The execution of tasks needs to be performed in a way that balances the work across execution contexts. The scheduler must ensure that all execution contexts are relatively well utilized. Without proper scheduling, the benefit of parallelization is diminished, since the execution is more sequential than it could be.

There are two conceptual approaches to scheduling tasks. Our framework uses both approaches, and we have implemented both. The first is to assign the forked tasks to a specific execution context, which we refer to as direct scheduling. The second is to let the execution contexts decide which tasks to execute, which we refer to as queue scheduling.

For both approaches, we assign some kind of priority to the tasks. The priority is based on where a task is in the fork hierarchy. We refer to the nesting level as the level of the task in the fork hierarchy, with the task at the top of the hierarchy at level one. The further down a task is, the higher its nesting level, and the less priority it has. Once a task reaches a certain nesting level, which we refer to as the maximum nesting level, all subsequent tasks forked from within that task are executed sequentially. The sequential execution reduces unnecessary overheads. Furthermore, no more tasks being forked, and therefore the sequence of traces does not have to be predicted because all execution is sequential.

5.4.1 Direct Scheduling

In direct scheduling, a task is assigned an execution context when it is forked. The task could be assigned to a different execution context, or to the same execution context as the fork. Three criteria are used to determine whether a fork assigns a task to the same execution context. A forked task executes on the same execution context if the fork is at a low enough level in the hierarchy, if the fork is the last one in the current task, or if
there is no more space on other execution contexts to execute the task\textsuperscript{13}. Otherwise, the fork is executed on a different execution context.

A disadvantage of direct scheduling is that there may not be enough information when a fork occurs to effectively schedule a task. Therefore, an assignment can be made that causes large gaps in the schedule. In our experience, if there is a mismatch between execution contexts and forks at each level in the hierarchy (e.g. 3 execution contexts and 2 forks per task with 2, 4, 8, etc. tasks at different levels), then the scheduling will cause one execution context to receive extra work that results in performance similar to having fewer execution contexts that are matched with the forks per task.

An advantage of direct scheduling is that it can be used to perform scheduling without using any centralized data structure to check which execution contexts are available. Instead of a centralized data structure, an algorithm can be used to predetermine which execution context should be used for each task. Our algorithm is based on an identifier that uniquely identifies an execution context and can be computed by each fork. The identifier is related to multiple subdivisions of the execution context count. Given $maxproc$ execution contexts and $breadth$ forks in a method at recursion depth $level$, the distance between each identifier is $\frac{maxproc}{breadth \times level}$ and is offset by the identifier of the method that contains the forks. The identifiers are mapped to execution contexts in a way such that contiguous execution contexts are used.

Direct scheduling is illustrated with the following example. Consider a system with 16 execution contexts and a task with two forks in it. The first task is executed on an execution context with identifier 0 and has a depth of 1. The distance between identifiers is $\frac{16}{2 \times 1} = 8$. Therefore, the first fork will execute on an execution context with identifier 8 and the second fork will continue executing on the execution context with identifier 0. Then at the second level, the distance between identifiers is 4. Thus, the first forked task

\textsuperscript{13}We assume that there is a finite number of resources (information stored per task such as inputs and outputs) that limit the number of tasks that can be assigned to a single execution context at any one point in time.
will result in forks to execution contexts with identifiers 8 and 12, while the second task will result in forks to execution contexts with identifiers 0 and 4. The process repeats until the distance between identifiers is 1, at which point all execution contexts are busy.

Figures 5.12 and 5.13 contain the scheduling that is performed for 16 execution contexts when the recursion breadth is 2 and 4 respectively. The execution contexts and identifiers are shown inside of each task invocation. This direct scheduling is useful because it allows work to be scheduled without having to read and write a centralized data structure.

5.4.2 Queue Scheduling

In queue scheduling, when a task is forked, it is inserted into a work queue. When an execution context has no work to perform, it chooses a task from the work queue. There are two points at which an execution context is idle: when it is not executing any tasks, and when it is waiting in a join for other tasks to complete. In both cases, a new task is taken from the work queue. The task that is chosen is the highest one in the level hierarchy. If there is more than one task at the highest level, then if the waiting is inside of a join, the first task that the join is waiting for is executed, and otherwise, the first
one in the queue is executed. This prioritization ensures that all the tasks that are being waited on execute, thus avoiding deadlocks. Furthermore, since the higher a task is in the level hierarchy, the more subtasks it will have, putting priority on tasks higher in the level hierarchy causes the tasks to be generated as quickly as possible, ensuring that the work queue fills up as quickly as possible, allowing for well balanced task execution.

Figure 5.14 contains an example of queue scheduling. Execution starts with task 1 on execution context 1. The task forks tasks 2 and 3, which are lower in the level hierarchy, and the tasks are inserted into the queue. At the same time, execution context 2 is checking the queue for work. Once task 2 is in the queue, execution context 2 takes it out and starts executing it. Then task 2 forks tasks 4 and 5. When task 1 reaches a join it retrieves task 3 from the queue and executes it. Although not shown, the execution would continue with task 3 spawning tasks, and then tasks 2 and 3 executing more tasks while waiting on joins. In this way, the execution contexts execute all of the tasks.
Figure 5.14: Executing tasks with a queue.
Chapter 6

Handling Dependences with Transactional Memory

Transactional memory enables the handling of dependences by optimistically executing transactions in parallel. The execution of the transactions is monitored for dependence violations. If a violation is found then a transaction is re-executed. Three challenges need to be addressed for transactional memory to be used to handle dependences of parallelized sequential programs. The challenges are to divide the program into sections that can execute in parallel, to execute these parallel sections efficiently, and to ensure that the execution is correct.

Parallel execution can be achieved by a good division of the parallelized program into transactions. If the transactions are too big then there is a higher likelihood of dependence violations, and running out of resources. If the transactions are too small then the overhead of starting and stopping them may reduce the benefit of parallelism. Also, if the point at which to start and stop transactions is chosen at runtime, then the algorithm for making that choice must be efficient. Otherwise the algorithm’s overhead will reduce the benefit of parallelism. Efficiency can be ensured by making any decisions at runtime quickly, and by avoiding the use of too many resources.
Only putting all work on tasks inside of transactions is not sufficient to ensure correctness. The reason is that there is no inherent order in which the transactions should execute. Therefore, for correctness to be ensured, not only must all the work be in transactions, but also the transactions must behave in the same way as if the transactions commit in sequential program order. When a transaction commits, the execution behaves as if that transaction executed atomically, i.e. in the same way it would execute if it executed by itself with no part of any other transaction executing in parallel. Thus, if transactions commit in sequential program order, the program is guaranteed to behave as if each transaction executed atomically in sequential program order. Therefore, the behaviour is identical to that of a sequential program.

The most straightforward way of ensuring that transactions behave in the same way as if the transactions commit in sequential program order is to make the transactions commit in this order. Our approach uses this straightforward way, and forces commits to occur in sequential program order. However, the approach can be extended without changing the underlying data structures to allow for more flexibility in the order of commits. The following sections contain a description, a couple of examples, and an overview of the limitations, benefits, and extension of our approach.

6.1 Approach to Sequential Program Commit Order

The following describes how trace-based parallelization and transactional memory can be used together in a way that meets the above challenges. Our approach requires two steps.

---

1 The range of behaviours of transactions can be thought of as varying from serializability to requiring a specific order. Serializable transactions only need to be isolated as if they execute one after another. There is no notion that one specific transaction has to occur before a different specific transaction, just that transactions occur one after another. Our approach requires that commits are in a fixed order. However, if an analysis is performed that shows that this order can be relaxed, then the behaviour can move more towards only serializability, while ensuring that execution still corresponds to the sequential program execution.
First, a relationship must be defined between tasks and transactions. Although every task could be made into a transaction, the overhead of starting and stopping a transaction for every task would be too large. Thus, we propose to use the nesting level of tasks to decide which tasks should be put in transactions\(^2\). Using the nesting level is good because decisions can be made quickly.

Tasks have transactions if their nesting level, described previously in Section 5.4, is between some specified minimum and maximum. All tasks that have a nesting level greater than or equal to the minimum nesting level and smaller than or equal to the maximum nesting level are referred to as transactional tasks, since these tasks can contain transactions. Using a minimum level may allow some tasks to not be in transactions, thus avoiding very large transactions, which may be problematic. Using a maximum level allows for avoiding the overhead of having too many transactions, since no new transactions are created at each nesting level beyond the maximum nesting level. This approach works well because creating the mapping between all the tasks and transactions is simplified to choosing two numbers. Thus, choosing when to start and end transactions can be done quickly.

An example is shown in Figure 6.1. The program on the left contains a number of tasks that each fork two other tasks and then perform some work. In the middle is an organization of the tasks into a tree, where minimum and maximum nesting levels are shown. Work is indicated by the letter \(w\). The size of the largest transactions can be controlled by the minimum nesting level. If it is more than one, then work performed in the tasks with a smaller nesting depth is outside of transactions. On the right, is the execution of the tasks on two execution contexts. Before the program can execute on multiple execution contexts, it needs to be transformed. The transformation results in the tasks being packaged in a way that allows them to execute on any processor, \(^2\)Tasks that are at the top of recursion may have dependences not handled, and therefore analysis may be needed to be performed to ensure execution is correct. Tasks at the bottom of recursion will be part of transactions at a higher recursion level, and therefore will have their dependences handled.
the control flow being modified to execute the tasks in parallel, and the transactions being added to the tasks. All execution that occurs outside of tasks is sequential. Tasks will execute on multiple execution contexts, with their work inside of transactions. The tasks will execute optimistically in parallel, and the transactional memory will serialize dependent execution.

In the figure, the work in the first task is not in a transaction. The benefit is that a potentially large transaction is avoided. However, there is no dependence handling outside of transactions, and if there are dependences between the first task and subsequent tasks then execution will be incorrect. The work of each task, indicated by $w$, at the second level is inside of a transaction\(^3\), and each task at the third level along with all the tasks that can be forked from that task are inside of a transaction. Thus, all work performed starting at the minimum level is inside of transactions.

Second, transactions must be made to commit in sequential program order. Transactional memory does not have a notion of sequential program order. Instead, transactions execute in any order that the transactional memory system chooses. The result must be the same as if each transaction executed atomically by itself. However, the order in

\(^3\)Each transaction is indicated by a dotted rectangle.
which the atomic execution occurs is arbitrary. Although it may be possible to change
a hardware transactional memory system to incorporate such a requirement, we propose
an alternative that does not require such change.

Transactions can be committed in sequential program order by making each trans-
action wait before committing until its predecessor transaction in sequential program
order commits. This is achieved by associating a variable with each transaction, and
telling each transaction which variable its predecessor is associated with. The variable
is initialized as unset, and is set when the corresponding transaction commits. The first
transaction in sequential commit order does not wait for any variable, and just commits
and sets its variable. Equivalently, the first transaction can be thought of as waiting for
a variable that is initially set by default. All subsequent transactions have to wait for
the variable of the preceding transaction to be set before they commit and set their own
variables. In this way, transactions can be made to commit in sequential program order.

The association of variables to transactions is non-trivial for arbitrary programs when
recursion may exist. The reason is that transactions are executed in parallel, and can
therefore start in an order different from the sequential program order. Indeed, a trans-
action could complete and be waiting to commit before a predecessor transaction starts.
Since waiting to commit requires knowing which variable a predecessor sets, the variables
cannot be associated directly with transactions at the start of each transaction.

Instead we keep track of variables based on when transactional tasks are forked, and
use this information to associate variables with transactions. Initially, each transactional
task is divided into consecutive phases that consist of either forking other transactional
tasks or performing work. Tasks at the maximum nesting level have a single phase and
do not fork transactional tasks. Then, each transactional task and each phase keeps track
of the variable that the first phase needs to wait on and the variable that the last phase
sets. The two variables are kept track of through references, which we refer to as firstRef
and lastRef respectively. In the figures in this chapter, these references are shown as first and last due to space constraints.

At the beginning of each phase, that phase’s firstRef and lastRef references are assigned. For the first phase, the firstRef reference corresponds to the firstRef reference of the task, and a new variable is created and the lastRef reference is made to point to this variable. For all subsequent phases, the firstRef reference is the lastRef reference of the preceding phase of the task. Also, for all subsequent phases except the last one, a new variable is created and the lastRef reference is made to point to this variable. For the last phase, the lastRef reference is the lastRef reference of the task. When a task executes its phases, new variables are created for all but the last phase. Since the new variables are created as necessary, the number of variables is not fixed, and parallel execution will not be stopped due to exceeding some predefined number of variables.

Each work phase is wrapped inside of a transaction that waits for the first variable before committing and sets the last variable. Each fork phase has its first and last variables passed to the forked task. Because the sequential program order of the phases is known, the variables are associated correctly within each task. Further, because the firstRef and lastRef references of a task are decided when a task is forked based on the sequential program order of the phases, the variables are associated correctly between tasks as well.

The general idea is to have a tree of tasks and to have references to variables for each phase of each task. For each phase, the references identify the variables that are important for the first and last transactions that will occur in sequential program order in that phase or any tasks below that phase in the tree. Because the phases in each task are executed in sequential program order, the references to each phase keep that order. The order is then passed down to the tasks and phases lower in the tree. If more tasks are forked, then the appropriate references are passed to them without affecting any other references in the tree. Thus, the transactions in the tree will always commit according to
the sequential program order specified by the references, and the references will always be correctly generated, regardless of the shape of the tree.

If transactions consist of fork phases followed by work phases, then in the absence of dependences the tasks can be executed completely in parallel. However, if work phases exist before fork phases, parallelism is limited. The reason is that whatever occurs after the work phase can only start after the commit of the transaction that the work phase is in. Therefore, the work phase and the phases that precede it can only execute sequentially with whatever occurs after the work phase. Parallelism only exists when there are multiple consecutive forks, which will execute in parallel. There is some flexibility in that the fork phases could be hoisted up above the work phase if no dependences are violated by the hoisting.

### 6.2 Examples

Two examples that show the same tasks being executed will illustrate how our approach works. In the first example, variables are created and named in sequential program order, with the first variable being $a$, the second being $b$, etc. In the second example, the variables are created and named in the order that the parallelized program executes in, and are named $x_1$, $x_2$, etc.

The first example is shown in Figure 6.2, which shows the complete execution of a set of tasks that contain recursion. The tasks are shown up to a nesting level of 3. The tasks are numbered 1 through 7 based on when they start in sequential program order, even though for arbitrary programs this order is not known ahead of time. The minimum nesting level is 1 and the maximum is 3. Each task consists of three phases, two fork phases followed by a work phase. For the tasks at the first two nesting levels, the figure contains a firstRef and lastRef reference (referred to as first and last due to space constraints), two forks, and a transaction that has a work phase. The transaction
waits for some variable and sets a different variable. At the maximum nesting level, each
task’s firstRef and lastRef references as well as transactions are shown. At the maximum
nesting level, the phases are shown combined by the text “...”. Control flow is shown in
solid lines while dependences on the firstRef and lastRef references are shown by dotted
arrows.

Figure 6.2: Executing tasks with sequential program order commits.

Task 1 starts executing with firstRef pointing to variable $a$, and lastRef pointing
to variable $h$. In this example the variables are ordered based on when tasks end in
sequential program order, even though for arbitrary programs the order is not known
ahead of time. Initially, the variable $a$ is set, while all other variables are unset. When
the first fork is encountered, a new variable $d$ is created and task 2 is forked. Task 2 has
firstRef pointing to $a$ and lastRef pointing to $d$. When the fork of task 5 is encountered,
a new variable, $g$, is created and task 5 is forked. Task 5 has firstRef pointing to $d$ and
lastRef pointing to $g$. Finally, work is done inside of task 1. This work must be inside
of a transaction to ensure that all dependences are handled. This transaction waits on
variable, $g$, and will eventually set the variable pointed to by lastRef of task 1, namely
variable $h$. Thus, for the first fork phase of task 1, firstRef points to $a$ and lastRef points
to $d$, for the second fork phase of task 1, firstRef points to $d$ and lastRef points to $g$, and
for the work phase of task 1, firstRef points to $g$ and lastRef points to $h$. The state of

---

4 In an implementation, set can correspond to 1 or true and unset to 0 or false.
the tasks at this point in execution is shown in Figure 6.3. Task 1 is executing and tasks 2 and 5 have been forked.

![Figure 6.3: Executing task 1 with sequential program order commits.](image)

Once task 2 is started, it forks tasks 3 and 4. When the fork of task 3 is encountered, a new variable $b$ is created and task 3 is forked. Task 3 has firstRef pointing to $a$ and lastRef pointing to $b$. When the fork of task 4 is encountered, a new variable, $c$, is created and task 4 is forked. Task 4 has firstRef pointing to $b$ and lastRef pointing to $c$. Finally, work is done inside of task 2. This work must be inside of a transaction to ensure that all dependences are handled. This transaction waits on variable, $c$, and will eventually set the variable pointed to by lastRef of task 2, namely variable $d$. The state of the tasks at this point in execution is shown in Figure 6.4. Tasks 1 and 2 are executing, and tasks 3, 4, and 5 have been forked.

![Figure 6.4: Executing task 1 and task 2 with sequential program order order commits.](image)

Tasks 3 and 4 are at a deep enough nesting level that they do not fork other tasks and all execution is performed on one execution context entirely inside of one transaction. The transaction in task 3 waits on variable $a$ and sets variable $b$. Since variable $a$ is set initially, task 3 can set variable $b$ without waiting for other tasks. The transaction in task
Chapter 6. Handling Dependences with Transactional Memory

4 waits on variable \( b \) and sets variable \( c \). Therefore, the transaction in task 4 must wait for the transaction in task 3 to complete. Thus, the two tasks can execute optimistically in parallel and then at commit time, the correctness can be ensured. The state of the tasks at this point in execution is shown in Figure 6.5. Tasks 1 through 4 are executing and task 5 is forked.

![Figure 6.5: Executing tasks 1 through 4 with sequential program order commits.](image)

Once the transaction in task 3 commits and sets \( b \), the transaction in task 4 can try to commit. If there are no dependence violations, the transaction will commit successfully. However, if there are any dependence violations this second transaction will abort and re-execute. Thus, dependences are properly dealt with, and the result is the same as if the program executed in sequential program order.

The transaction in task 2 can execute in parallel with the transactions in tasks 3 and 4. Once variable \( c \) is set in task 4 the transaction in task 2 can then commit and set \( d \). Tasks 2, 3, and 4 can optimistically execute in parallel with task 5 and any tasks it forks.

Task 5, which has firstRef pointing to \( d \) and lastRef pointing to \( g \), forks tasks 6 and 7 and then performs work. A new variable, \( e \), is created for task 6, which has firstRef pointing to \( d \) and lastRef pointing to \( e \). When the fork of task 7 is encountered, a new variable, \( f \), is created. Task 7 has firstRef pointing to \( e \) and lastRef pointing to \( f \). Finally, work is done inside of task 5. This work must be inside of a transaction. This transaction waits on variable, \( f \), and will eventually set the variable pointed to by lastRef of task 5, namely variable \( g \). The state of the tasks at this point in execution is shown.
in Figure 6.6. Tasks 2, 3, and 4 have completed executing, tasks 1 and 5 are executing, and tasks 6 and 7 are forked.

Figure 6.6: Executing tasks 1 through 5 with sequential program order commits.

Tasks 6 and 7 are at a deep enough nesting level that they do not fork other tasks and all execution is performed on one execution context entirely inside of one transaction. The transaction in task 6 waits on variable $d$ and sets variable $e$. The transaction in task 7 waits on variable $e$ and sets variable $f$. Therefore, the transaction in task 7 must wait for the transaction in task 6 to complete. The two tasks can execute optimistically in parallel and correctness is checked at commit time.

Furthermore, task 6 needs to wait on variable $d$, which is inside of task 2. This behaviour ensures correct execution. The two tasks have different parent tasks and there is no need for the tasks to know of each other. The only important factor is whether the appropriate variable is set or unset. Therefore, the commit order is preserved across tasks in any part of the task hierarchy, with the only information required being which variables to wait for and set.

Once the transaction on task 7 commits, the transaction on task 2 can commit, after which the transaction on task 1 can commit. At the end all transactions will have executed, and all work that the tasks need to perform will have been executed in the same way it would have executed in sequential program order. This final state is shown at the beginning of this example in Figure 6.2. When there are no dependences, then all execution can be in parallel. If dependences exist then some transactions will be re-
executed. Thus, even though the execution can be out of order, the program will execute correctly.

Although in the above example the variables are in the order of sequential program commits, in reality they would be in the order that they are created. Therefore, the second example shows variables being created and named in the order that the parallelized program executes. Figure 6.7 contains the same diagram as the previous example, except that the variables are based on when they are created. For simplicity, each variable is $x$ with a numbered subscript indicating the order in which the variables are created. The figure only contains the numbered subscript to conserve space. When task 1 is executing, it has firstRef and lastRef pointing to variables $x_1$ and $x_2$, task 2 has firstRef pointing to $x_1$ and lastRef pointing to $x_3$, and task 5 has firstRef pointing to $x_3$ and lastRef pointing to $x_4$. Then, assuming that tasks 2 and 5 execute in parallel, with task 2 executing slightly earlier, firstRef and lastRef references are assigned for tasks 3, 4, 6, and 7. Task 3 has firstRef pointing to $x_1$ and lastRef pointing to $x_5$. Task 4 has firstRef pointing to $x_5$ and lastRef pointing to $x_6$. Task 6 has firstRef pointing to $x_3$ and lastRef pointing to $x_7$. Task 7 has firstRef pointing to $x_7$ and lastRef pointing to $x_8$. The variables are then set by the tasks in the order $x_5$, $x_6$, $x_3$, $x_7$, $x_8$, $x_4$, and $x_2$.

Figure 6.7: Executing tasks with transactions when variables are ordered by when they are created.

This approach allows for additional tasks to be added without affecting the other tasks. For example, assuming that the maximum nesting level is larger, task 6 could fork
two tasks, tasks 8 and 9, with firstRef and lastRef pointing to variables $x_3$ and $x_9$ for task 8, and $x_9$ and $x_{10}$ for task 9. No other tasks need to be changed. Furthermore, task 8 will wait for the transaction in task 2 to commit, even though task 2 does not have information about task 8 and task 8 has no information about task 2. Therefore, using our approach allows arbitrary tasks to be executed without having to know what they are ahead of time.

### 6.3 Limits, Benefits, and Extension

Our approach to ensuring that commits are performed in sequential program order has two limitations. First, the current implementation requires a transactional memory system that allows for instructions to escape transactions and has lazy version management and lazy conflict detection. The escape mechanism is required because the waiting on a variable would otherwise cause a conflict, causing unnecessary and possibly problematic transaction aborts. The lazy version management and conflict detection are beneficial because otherwise transactions could abort early and in the wrong order, possibly leading to livelock. Second, dependences that always exist, such as dependences on induction and reduction variables will completely serialize execution. Therefore, to obtain the best results, transactional memory has to be combined with dependence removal.

Our approach to ensuring that commits are performed in sequential program order has four benefits. First, it can work without any modification to the transactional memory system. All the information can be maintained by software and is independent of the transactional memory system. Second, the amount of information that needs to be stored is small, since for each transaction one variable and two references need to be kept track of. Third, the variables can be assigned arbitrarily. Thus, a pool of variables can be used, and a variable can be drawn from that pool whenever a fork is encountered, regardless of how many tasks will be executed. Therefore, any arbitrary set of tasks can be executed
effectively by a hardware transactional memory system and the tasks will commit in sequential program order, ensuring correct execution.

Fourth, this approach can be extended easily to relax the ordering of transactions and avoid a full sequential ordering of transaction commits. For example, if an analysis can determine that two tasks do not have dependences then their commits do not have to be ordered. All that is needed is to adjust the firstRef and lastRef references appropriately.
Chapter 7

Implementation

Implementing our parallelization framework gives rise to many non-trivial issues that require solutions of considerable complexity. This chapter focuses on these issues and our solutions to overcome them. Issues exist for each of the four components and for using HTM.

The main issue for the grouping component is creating the graph of traces based on a representation that is different from the representation used for traces. The main issue for the packaging component is only being able to compile one method at a time. The main issues for the dependence component are how to identify dependences efficiently and how to properly insert joins. The main issue for the scheduler is making it efficient. The main issue for using HTM is dealing with the conflicts that arise between actions performed by a JVM and transactions.

We have implemented two prototypes of our framework. Both prototypes are offline feedback directed systems that can use traces to parallelize sequential programs. One of the prototypes parallelizes programs and uses HTM to deal with dependences. This prototype uses LogTM and runs on top of a simulated system. To overcome the limitation of evaluating only on a simulated system, we have created a prototype that automatically parallelizes programs on a real system. Since HTM is not used, only dependences
on induction and reduction variables are handled. This prototype is an extension of the Jikes RVM [42]. We refer to the prototypes as the LogTM prototype and Jikes prototype. The remainder of this chapter contains the overview of the Jikes prototype, how we addressed the issues for the four components, a discussion of the issues that we encountered when using Java Virtual Machines on an HTM system, and a description of the LogTM prototype.

7.1 The Jikes Prototype

The Jikes RVM is a compiler based JVM that contains a baseline compiler and an optimizing compiler. All Java methods in a program are compiled using the baseline compiler, and when certain methods are executed more frequently, they are optimized by the optimizing compiler. In the preliminary execution, a trace collection system monitors the program and records traces. The baseline compiler is modified to insert the appropriate monitoring instructions that tell the trace collection system which instructions are being executed. The primary execution consists of transforming the program to be parallel and executing the parallel version. The transformation is performed when methods are compiled using the optimizing compiler in the Jikes RVM. The transformation occurs during the primary execution of the program because the Jikes RVM compiles methods at runtime.

We have modified the optimizing compiler in the Jikes RVM to group traces into tasks, extract and package tasks, and then to insert instructions that handle dependences. A scheduler has also been added that allows the tasks to execute in parallel.

The scheduler distributes the tasks to the execution contexts. For the Jikes prototype, the contexts correspond to the main thread and a set of worker threads, where each thread is assigned to a specific processor on the computer that is executing the program.
The instructions that handle dependences only handle a limited number of dependences. Induction and reduction variables are identified and their dependences removed. Furthermore, joins are inserted for the corresponding forks. After the optimizing compiler is done, the program is parallelized and the traces grouped into tasks can execute in parallel. This prototype performs automatic parallelization of sequential programs, and allows traces on these programs to execute in parallel. However, the prototype does not handle dependences beyond induction and reduction variables\(^1\). Therefore, the prototype is effective at measuring the overheads of our framework and the performance that can be achieved when executing programs with simple dependences. To evaluate programs with more complex dependences, these dependences need to be handled, for example through an HTM system. The LogTM prototype addresses this shortcoming and is described in Section 7.7.

### 7.2 Grouping Component

The grouping component takes traces generated by the trace collection system, and generates SCCs and tasks from a graph of traces. Therefore, before SCCs can be generated, a graph of traces needs to be created. The trace collection system collects traces at the bytecode level. However, the Jikes RVM’s optimizing compiler has a different representation of the program which is an intraprocedural CFG\(^2\). Therefore, a mapping needs to be created between the bytecode of the traces and the representation used in the optimizing compiler, which is referred to as intermediate representation (IR)\(^3\).

---

\(^1\) Other dependences are ignored.

\(^2\) A CFG is generated for each method that is optimized. Each CFG indicates the possible control flow, and can always be generated. Each CFG only represents part of a program, and therefore does not have to indicate all control flow.

\(^3\) We have also grouped traces using bytecodes. The mapping is not required for such an approach to grouping. However, this approach requires that edges between traces are recorded along with traces by the trace collection system. We have decided not to use this approach because it does not take into account what is being compiled to identify what the tasks should be.
The mapping between traces and the IR of the optimizing compiler is made part of the creation of the graph of traces. Thus, generating SCCs and tasks are independent of the underlying implementation specific representation.

Although one possibility is to use the IR to collect traces, we have chosen not to. The reason is to have the offline system be as similar as possible to an online system. In an online system, the baseline compiler generates the methods initially, and the optimizing compiler is only used afterwards. Therefore, the baseline compiler needs to add the appropriate monitoring instructions that interface to the trace collection system. Since the baseline compiler does not have information regarding the IR, the traces have to be collected without this information.

The following sections describe our approach to the mapping of traces to the IR and the identification of edges between traces. The difficulties that our approach needs to overcome are that recursion needs to be taken into account and that traces, which can span methods, and basic blocks in the IR have different boundaries.

### 7.2.1 Mapping Traces to the Optimizing Compiler’s Control Flow Graph

The input of the mapping is a set of traces each with a sequence of basic blocks, which are referred to as *trace blocks*. The output for each trace is four sets of basic blocks in the optimizing compiler’s CFG, referred to as *IR blocks*. The first two sets contain information about trace starts and the second two sets contain possible control flow at trace exits. This information is used to find edges between traces. The reason for having two sets for each type of information is the need to differentiate between the beginning and middle of *IR blocks*. The first set is the *startAtStart* set, which contains the *IR block* that corresponds to the trace start, if the trace starts at the beginning of an *IR block*. The second set is the *startInMiddle* set, which contains the *IR block* that corresponds to the trace start, if the trace starts in the middle of an *IR block*. An example of a trace starting
in the middle of an *IR block* is a trace that starts at the target of a return, which is the instruction after an invoke. Although that instruction would start a new basic block in an interprocedural CFG, it is not at the start of a basic block in an intraprocedural CFG. One of these two sets will have one *IR block* and the other set will be empty. The third set is the *inTrace* set, which contains *IR blocks* that are in the trace. The fourth set is the *successors* set, which contains the *IR blocks* that are targets of the *IR blocks* in the trace. Once all the sets are filled, the relationships between the *IR blocks* in the sets of different traces is used to generate edges between traces.

The sets are created in two parts. First, all the sets are created without considering recursion. Second, the sets are added to by considering recursion. Both parts need to be performed to capture parallelism in recursive and iterative programs.

The first part is shown in Algorithm 2. The algorithm uses certain information that is part of the optimizing compiler’s CFG. The following is a description of the terms that are used in the algorithm. Each *IR block* contains a sequence of instructions, *instructions*, a set of targets, *targets*, and a number, *number*, that represents the location of the block in code order. Furthermore, each block has a flag, *hasInlineGuardPatchPoint*, that indicates if the block contains an inline guard patch point, which means that the block ends in a test to see if the inlining of a method is valid or not. The test will determine whether the inlined code, which is part of the optimizing compiler’s CFG, should execute or whether the called method should be invoked. Finally, each instruction has information about what type of instruction it is and information specific to the type of instruction. In particular, the algorithm needs to identify call instructions and the methods they call. The information that identifies whether an instruction is a call is referred to as *isCall*. The information regarding which method is called is referred to as *targetMethod*. Similarly, the optimizing compiler’s CFG has information regarding the method that it represents. This information is referred to as *thisMethod*. 
Algorithm 2 Algorithm to map trace blocks to sets of IR blocks.

for each trace, t {
    beenAdded=false
    for each appropriate trace block in t, tb {
        for each IR block, cb {
            for each instruction in cb.instructions, i {
                if (i in tb) {
                    // add initial
                    if (!beenAdded) {
                        if (i is first in cb)
                            t.startAtStart.add(cb)
                        else
                            t.startInMiddle.add(cb)
                        beenAdded=true
                    }
                    // add all including fallthroughs
                    stack.push(cb)
                    while(!stack.empty) {
                        b=stack.pop
                        if (t.inTrace.contains(b))
                            continue
                        else
                            t.inTrace.add(b)
                            addAll=b.hasInlineGuardPatchPoint
                            for each block in b.targets, target {
                                if (addAll)
                                    stack.push(target)
                                    t.successors.add(target)
                            }
                            if (!addAll && |b.targets|==1 && b.number > target.number)
                                stack.push(b)
                            break // ignore remaining instructions on this block
                        }
                    }
                }
            }
        }
    }
    if (!beenAdded)
        break
}
The algorithm works by processing each appropriate trace block on each trace. The trace blocks need to be chosen carefully because there may be a mismatch between the methods on the trace and the method being compiled by the optimizing compiler. For example, a trace can start on one method and then through a call or return go to a different method. If the method that is being compiled appears only later in the trace, then the trace start would not be identified correctly. To avoid this problem, only a subset of trace blocks is processed. Although there are many ways to choose the subset, we have decided to use those trace blocks that are on the top method of the trace, and leave other ways as future work. A top method is the earliest method on the call stack out of all the methods on a trace \[9\]. When recursion occurs, all blocks in the top method are considered, even if they occur at different levels of the call stack. This approach ensures that all trace blocks on the top method of a trace are considered. The situation for which this approach is useful is the compilation of a recursive method, \(a()\), when a trace is recorded starting near the end of a different method, \(b()\), that is called from \(a()\). For example, if the trace contains a return from \(b()\) to \(a()\), and then a return from \(a()\) to \(a()\), then using this approach will allow the trace to be used while compiling \(a()\). We have found focusing on top methods useful, because they allow a larger amount of execution to be captured, and thus potentially parallelized.

For each trace block, \(tb\), on each trace, the instructions on all IR blocks are examined. If any instruction is on the trace block \(tb\), then that IR block is processed. If no match is found for the first trace block being considered, then the processing is aborted. That ensures that traces that do not start properly will not be considered for grouping. The processing consists of two steps.

The first step occurs only for the first trace block of each trace. This step consists of adding the IR block either to the startAtStart or startInMiddle sets, depending on whether the instruction is the first instruction in the IR block or not. For each trace, only one of these sets will be non-empty, and that set will have one element.
Chapter 7. Implementation

The second step, which occurs for all \( IR \) blocks, consists of filling in the inTrace and successors sets of each trace. The \( IR \) block being processed is put on a stack, and processing of each \( IR \) block on the stack continues until the stack is empty. If an \( IR \) block is already in the inTrace set, then nothing is done. Otherwise, the \( IR \) block is added to the set. Then, all the \( IR \) block’s targets are added to the successors set. If the \( IR \) block is an inline guard patch point, then all of its successors are added to the stack. The reason is that inlining may occur that causes straight line code to be turned into two paths of execution, one for the inlined method and one for the invocation of the method in case inlining is incorrect. Treating all the blocks as part of the trace allows for the processing of traces that contain inlined methods. Then, if the \( IR \) block has only one target that has not been put on the stack and is further in the code (i.e. the \( IR \) block does not end in a backward branch), then that target is added to the stack. The addition of this target captures traces that contain straight line code that spans multiple \( IR \) blocks because of control flow. One example of such a scenario is branch initialization and test code that is divided between \( IR \) blocks because of a backward branch to the test. The end result is the filling of the appropriate sets for each trace that allow edges between traces to be formed.

The second part of creating the sets consists of identifying successors arising from recursive code and is shown in Algorithm 3. The algorithm goes through all \( IR \) blocks in each trace, and if a recursive call is found, then the start of the method is added as a successor of the trace. Although other calls may result in executing methods that contain traces, the calls are treated as single instructions to be executed. Since the called methods are not inlined into the current method, any traces on these methods would have to be extracted and packaged when those methods are optimized. The \textit{same} method is true if the name and descriptor of the methods match, which allows recursion for virtual methods to be captured as well.
Algorithm 3 Algorithm that considers recursion for mapping trace blocks to sets of IR blocks.

for each trace, t {
    for each block in t.inTrace, b {
        for each instruction in b, i {
            if (i.isCall && same(i.targetMethod, thisMethod))
                t.successors.add(thisMethod.firstBlock)
        }
    }
}

7.2.2 Identifying Edges between Traces

Once the four sets for each trace are identified, edges can be created. Identifying edges is complicated by starting traces in the middle of IR blocks. When there is an edge to a trace that starts at the beginning of an IR block, that edge is added to the graph of traces. However, under certain circumstances, an edge to the start of a trace in the middle of an IR block should not be added. Algorithm 4 describes how the edges are created. For each pair of traces, an edge is added to the graph, g, from the source trace to the target trace when the start IR block of the target trace is a successor of the source trace. The condition to determine whether the edge should be added is different depending on if the target starts in the middle of a block or at the start.

Algorithm 4 Algorithm that generates edges for the graph of traces.

for each trace, source {
    for each trace, target {
        for each block in target.startInMiddle, b { // only one iteration
            if (source.successors.contains(b)) {
                if (source.inTrace.contains(b) || b.number==0 || source.number!=b.number)
                    g.addEdge(source, target)
            }
        }
        for each block in target.startAtStart, b { // only one iteration
            if (source.successors.contains(b))
                g.addEdge(source, target)
        }
    }
}
If the target starts in the middle of a block, then the condition is met if that block is also part of the source trace, if the block is the start of a method, or if the source and target start on different blocks. The first condition corresponds to an edge that exists between two traces where the source trace has an IR block that has a call to some method, and the target trace starts after the method returns. The second condition corresponds to an edge at a recursive call. The third condition corresponds to other possibilities, such as a branch, where the target starts in the middle of a block because the beginning has instructions that are part of the IR and are not directly related to bytecode. These instructions do not have proper bytecode indices. The block numbers must be different to avoid spurious self edges. Otherwise a self edge would be added when a trace starts in the middle of a block and has that block as a successor. Such a trace does not have a self edge because the trace does not start at the beginning of the block. Therefore, no edge should be added. Although some self edges that should exist may be missed, that has not been a problem in our experience.

If the target starts at the start of a block, then testing the condition is simpler. The reason is that the relationship between blocks is clear cut in that the control flow is part of the same method and there is a clear relationship from one block to the very beginning of another block. The condition is met when the block is one of the successors of the source trace.

Once the graph of traces with all the edges is generated, the SCC and task generation can proceed on the graph as described in Section 5.1\(^4\). The underlying difficulties of mapping between bytecode and the IR are dealt with in graph generation, separately from

\(^4\)All parts are implemented except the test to determine which task a trace is in when tasks have overlapping starts and ends. The reason is that the benchmarks used in the experimental evaluation do not have such overlaps. Therefore, just the starts and ends are used to identify traces on tasks. The main reason to have tasks with overlapping starts and ends is to have tasks at multiple levels of a loop nest with many instructions at each level before and after the corresponding next nested loop, where tasks encompass all instructions at individual levels. To deal with such cases, an extra pass over the control flow could be performed that propagates information regarding the starts and ends of tasks to successor basic blocks. Since our initial implementation works properly for the benchmarks that we use, we do not implement this additional step.
the SCC and task generation algorithms. Thus, the SCC and task generation algorithms can continue to be generic and independent of the underlying implementation.

The tasks that are produced by the grouping component contain the IR blocks of the traces. The IR blocks in the tasks are kept as is, without changing the compiler’s CFG. In particular, no transformation is performed to change the CFG in a way that separates traces so that each trace has its own basic blocks and can be optimized. Therefore, conceptually the tasks execute the traces in parallel, even though there is no direct relationship between an instruction in a task and a specific trace. Rather, an instruction could be part of multiple traces. We take this approach to ensure that the sequential and parallel versions of the program are as similar as possible.

Furthermore, because the Jikes prototype does not fully deal with dependences, the tasks have to be modified to remove control flow dependences. That has been achieved by expanding tasks. A task that has a set of IR blocks is expanded to contain all IR blocks between the first and last IR block in that set of IR blocks. This approach works well in practice, and is necessary if dependences are not fully handled.

7.3 Packaging Component

Extraction and packaging require that tasks are in their own methods and all data is properly passed between tasks. To achieve these goals, outlining of the tasks needs to be performed. Outlining is the moving of part of a function or method outside of the main part of the function or method [56]. Usually, infrequently executed code is outlined to not affect the frequently executed code. In contrast, we outline frequently executed code into its own method to enable this code to be executed on multiple processors. For tasks that are parts of loops, outlining is necessary, while for tasks that are parts of recursive methods, outlining of the slow version of a task allows for a clean separation between the slow and fast versions of the task (see Section 5.2).
When outlining is performed on a method, part of this method is put into a new method and replaced with instructions that call this new method and deal with what this new method returns. We refer to the method that initially contains the part to be outlined as the original method, and we refer to the method that contains the part to be outlined at the end of outlining as the new method.

Performing outlining on part of a Java program at runtime requires overcoming three issues. The first issue is that a runtime Java compiler can only process one method at any one time, while outlining requires that two methods are processed. Outlining requires that the original and new methods are processed together, because the instructions that are removed from the original method have to be the same as the instructions that are inserted into the new method. Therefore, the compiler needs to analyze and keep track of instructions in both methods. However, the runtime Java compiler in the Jikes RVM is designed to process methods one at a time. One approach to overcome this problem would be to modify the compiler to compile multiple methods at the same time, which would require that the compiler is changed to process multiple methods and to keep track of multiple CFGs at the same time. An approach that requires fewer changes to the compiler is to divide the outlining into two parts and explicitly pass information between them. We take this latter approach since we prefer to make as few changes to the compiler as possible.

The second issue is the difficulty of sharing information between these two parts of outlining, especially the information regarding instructions. The third issue is that of efficient information passing to and from the outlined code. The main source of overhead is creating the space for the information\(^5\). Therefore, we preallocate and reuse the space.

\(^5\)In a naive implementation, the overhead arises due to several factors. The first factor is memory management overhead. Every time a method is called, new memory needs to be allocated. The second factor is that when the outlined code is called many times, a large amount of memory is allocated. Thus, the number of garbage collections required increases.
7.3.1 Two-Part Outlining

We divide outlining into two parts. The first part occurs when the original method is compiled, and corresponds to steps 1, 4, and 5 of the six step extraction and packaging process in Section 5.2: blocks and exits are identified, the first block of a task is replaced, and instructions to save and restore variable values are added to the original method. The second part, which occurs when the new method is compiled, corresponds to steps 2, 3, and 6: the new method’s blocks are created, entry and exit blocks are added, and instructions to save and restore variable values are added to the new method.

Two problems are caused by the new method being created after the original method is compiled. First, the original method cannot be made to call a method that does not exist. Second, no mechanism exists to create the new method. To solve these problems, we make the original method call a lazy compilation method, which compiles the new method, changes the call to the lazy compilation method into a call to the new method, and then calls the new method. Thus, both problems are solved: the new method is created, and the original method can call the new method.

7.3.2 Instruction Matching

In the Jikes RVM, methods are compiled one at a time, and they cannot share intermediate representation data such as instructions and variables. Thus, a method that contains packaged tasks cannot have a copy of the instructions in the corresponding original method. Instead, the instructions have to be regenerated from the bytecode of the original method\(^6\). Once the instructions are generated, a mapping is created that links instructions in a new method to those in the original method. This mapping is necessary to properly identify basic blocks on tasks and pass variables between the original and new method. A simple one-to-one mapping is used that assumes that the instructions gener-

\(^6\)Although all instructions in a method are initially generated from bytecode, the ones that are not on the appropriate tasks are removed.
ated from the same bytecode are the same for any compiled method. For this one-to-one 
mapping to be sufficient, inlining and class loading need to be addressed\textsuperscript{7}.

Inlining decisions made by the Jikes RVM affect the instructions in a method and are 
based on profiling information that is collected as the program executes. This information 
consists of a partial weighted call graph that the Jikes RVM maintains. The weights are 
based on how frequently a call site executes, and will decrease over time when a call site 
is not executed. If a call site has a large weight then there is a higher probability of it 
being inlined. Since new methods are compiled at a different time from the original, their 
compilation will use different profile information, and there is a possibility that different 
inlining decisions will be made. To prevent this possibility, the profile information used 
for compiling the original method is saved and used when compiling all related new 
methods.

Class loading affects the instructions because a method that references a class that 
is not yet loaded will contain placeholder instructions instead of instructions that access 
the class. Therefore, instructions generated before this class is loaded will be different 
from those generated after this class is loaded. Although the mapping would need to be 
more robust to avoid this problem, this problem does not occur in the normal operation 
of our prototype, and thus we leave creating a more robust mapping for future work.

### 7.3.3 Variable Passing

The passing of variable values between the original methods that have tasks extracted 
from them and those new methods that contain packaged tasks must be efficient, al-
low multiple variables to be written back, and work with methods that are part of the

\textsuperscript{7}Our prototype transforms a method soon after the initial intermediate representation CFG is gener-
ated. Therefore, most optimizations are performed afterwards and the prototype is not affected by them. 
Also, most optimizations only use the CFG of the method as input, and therefore when the same CFG 
is given the same output is produced. Thus, the instruction mapping is not affected by optimizations. 
The only exceptions arise due to having input other than the CFG, which occurs due to class loading 
and inlining.
scheduler. For tasks that encompass entire recursive methods, the method structure is sufficient. However, for tasks that are only parts of loops, there is no underlying structure to use, and therefore a different approach needs to be taken.

We use arrays to pass the variable values. Using arrays allows for passing variables to be performed efficiently and uniformly. The arrays for each new method are allocated when the method is created. The instructions that save the values store them into the arrays, and the instructions that restore the values read from the arrays. The references of the arrays are passed in as parameters to the new methods and to the scheduler. Therefore, the parameters always have the same types, those of the arrays being used. This design is chosen because writing to preallocated arrays is relatively efficient, the new method can write directly to the arrays, and all new methods have identical parameters.

Our prototype has to also deal with guard variables that are used for runtime error checks in the Jikes RVM and cannot be passed between methods. The guard variables that are read by instructions in the new method are initialized to a default value at the beginning of the new method. Similarly, the guard variables that are written inside of the new method are set to default values in the original method. Although a more robust solution would be desirable, this approach does not affect the execution of the applications used in the experimental evaluation. Thus, we leave the implementation of such a solution as future work.

---

8Three types of arrays are created: object arrays, integer arrays, and double arrays. These three types of arrays correspond to the three types of data structures used to transfer data via Jikes RVM’s reflection mechanism, which is what our initial attempts used (reflection turned out to be too slow). Object references are manipulated directly in the object arrays via IR instructions, thus avoiding having to worry about casting. By having the three different types of arrays, boxing/unboxing issues are also avoided.

9The Jikes RVM is designed to have each guard variable be tied to a specific compiled method, and trying to pass the guard variables and then using them in a different compiled method causes crashes in the Jikes RVM.
7.4 Dependence Component

Once the tasks are extracted and packaged, dependences need to be dealt with. The dependence component in our Jikes RVM prototype can deal with reduction and induction, as well as dependences from forks that require joins.

7.4.1 Reductions and Inductions

To be able to deal with induction and reduction variables automatically, all dependences need to be identified based on whether they are a combination of local accesses, field accesses, static accesses, array accesses, inductions, and reductions. To identify which variables can be changed to reductions and inductions data flow analysis needs to be performed. The analysis consists of two steps.

In the first step, reaching definitions, use-def chains, dominators, and available variables are computed. These are computed based on standard approaches [57], although there are three differences. First, the analysis is performed on only those parts of a method that contains tasks instead of on the entire method. The analysis is performed on each task separately. If induction and reduction variables are found by this analysis, then the task keeps track of them. Second, the standard approach of using sets backed by hashtables is too slow. Although we could have tried using bit vectors, we have decided to use a set data structure that contains a reference variable and an optional hashtable. This data structure allows for very fast set operations for sets of zero or one variables, which tends to be the common case, at the expense of slightly slower performance for larger sets. The resulting performance is fast enough to perform the computations at runtime with little overhead.
In the second step, data accesses are analyzed. All local variables are analyzed\textsuperscript{10}. Uses of a local variable not in the live variable set and with dominating definitions do not cause dependences. Other uses have potential dependences. The potential dependences are then analyzed and if they fall into the patterns of reductions or inductions, then the appropriate code is generated to deal with them and to provide enough information to the scheduler to be able to find the values that the induction variables will have. All remaining dependences need to be dealt with in a different manner, such as speculation.

Our current implementation handles inductions that go between some starting and ending bound in fixed increments and reductions that are simple additions, subtractions, multiplications, divisions, less than comparisons, less than or equal comparisons, greater than comparisons, and greater than or equal comparisons. The comparisons are used to compute the maximum and minimum values of a sequence of elements. The implementation could be extended to include other types of inductions and reductions, although that is left for future work.

\subsection{Join Insertion}

When a task is forked, one or more corresponding joins need to be inserted. These joins have complex logic that allows them to ensure that threads will not be idle when work is available. If there are tasks in the work queue, and a join is waiting for another task to complete, then the join will execute tasks from the work queue. For the case when a task is associated with an induction variable, the scheduler will use the information about the induction variable to identify all the corresponding dynamic tasks and distribute them across execution contexts. Therefore, in this case, a simple join will wait for all tasks to complete, and does not need complex logic. The following is a description of how the complex joins operate and are inserted.

\textsuperscript{10}Heap accesses could be analyzed as well. The implementation has the ability to identify array, field, and static accesses. However, since no alias analysis is performed, too many false aliases would occur, and this functionality is not used.
A join that is inserted for a particular fork consists of waiting for that fork to complete and write the variable. While the waiting is being performed, a different task may be executed based on tasks that are waiting to execute, and based on a task’s location in a task hierarchy, which is maintained as a tree of tasks. Therefore, the join waits for a forked task to complete, and the join potentially executes other tasks while waiting. When the forked task completes, the join will read the appropriate data from the forked task, and then write that data to the appropriate variable. If the fork happens to write to an array element, then the index of the element needs to be saved by the fork for the join. Thus, the end result is that the joins effectively execute all tasks on the work queue.

When a task is forked, a corresponding join needs to be inserted before any data dependences are violated. Currently, the only dependence that is dealt with is on the return value. Therefore, a join must be inserted before the return value is used. Since finding the best points to insert the join is difficult, an easier approach is taken. Joins are inserted before every use of the return value. Furthermore, since the compiler optimizes one method at a time, the compiler cannot know what other methods can do. Therefore, joins are inserted before all calls to other methods\(^\text{11}\) and before all returns. Also, joins are inserted after loop nests to ensure that tasks from multiple loops are not executed together. The reason is that there may be dependences between tasks from different loops which would not be detected due to not having a full dependence analysis.

When the first join that corresponds to a fork is encountered, that join waits for the fork and ensures that the returned value is stored correctly. Subsequent joins that are encountered perform no work, and thus have very little overhead. Therefore, although there is some overhead in having many joins for each fork, the overhead is small\(^\text{12}\).

\(^{11}\)The optimizing compiler has already performed inlining before our framework transforms the code. Thus, there is no issue with having unnecessary joins before small methods.

\(^{12}\)Although we did not measure exact overheads of the joins, if the overheads were large then our experimental evaluation would show significant overheads. Since the overheads shown in Section 8.7.2 are small, we believe that the overheads of joins are acceptable.
If there are joins in the same block, forks and their corresponding instructions are moved to the beginning of the block and joins are moved to the end as much as possible. Also, having a large number of tasks in the work queue can lead to excessive resource usage. To keep the resource usage down, the resources used by tasks that have already completed are made available before each new task is forked. This freeing of resources is achieved by inserting a non-bound join before forks. This non-bound join checks if there are any tasks that have completed their work but have not saved the results because the corresponding join has not been called. If there are any found, the data is stored, thus making the resources used by that task available.

7.5 Scheduler

The scheduler ensures that the tasks are scheduled and the threads execute these tasks. Each worker thread waits until tasks are forked for it to execute. The thread will then execute these tasks. The main thread executes the sequential part of a program until a task is encountered. The task is executed and more tasks are potentially forked. When a task is forked, the scheduler decides where it should execute. The tasks can be scheduled in one of two ways: either tasks are assigned to threads directly or threads decide which tasks to execute (see Section 5.4).

If tasks are assigned to threads directly, then when a task is forked the scheduler will choose a thread for the task to execute on and inform that thread. If the chosen thread is a different thread from the one that the fork is on, then this other thread is notified and will execute the task. If the chosen thread is the same thread, then it will execute the task immediately.

If threads decide which tasks to execute, then when a task is forked, the scheduler will insert the tasks into the work queue and maintain information about these tasks.

\footnote{When a thread waits, it will yield to allow other threads that are part of the JVM to give them a chance to execute.}
Threads will execute tasks if they are not busy executing a task. A thread is not busy when it has no tasks to execute, or when it is executing a join that is waiting on some task. When a thread is not busy then it will get the most appropriate task from the work queue and execute it. If no tasks can be found, then the thread yields to other threads.

Furthermore, when a task is forked that has one or more induction variables, which we refer to as a *loop-based task*, the scheduler uses the information to fork multiple tasks. Each task will be given different values of the induction variables and the tasks will be combined into as many groups as there are threads. Then, the scheduler distributes the groups of tasks to the worker threads and executes one of the groups of tasks on the main thread. The benefit is that unnecessary overhead is avoided.

Various approaches can be used to distribute the induction variable values. The simplest is to divide the values into equally sized sequences (e.g. for an induction variable going from 0 to 99 and two threads, divide the values into sequences from 0 to 49 and from 50 to 99). Each thread is then given one of these sequences. This approach is taken in the experimental evaluation, although other approaches could be used. The prototype can use other approaches that assign multiple sequences to each thread (e.g. 0 to 24 and 50 to 74 to the first thread and 25 to 49 and 75 to 99 to the second thread).

Further, approaches may be used that consider dependences between loop iterations. There may be dependences between instructions in different iterations of the loop. These dependences will limit the amount of available parallelism. If the groups of tasks that are scheduled together are too large, then the execution may be forced to be sequential. However, if the induction variable values are divided properly and a good size is chosen for the groups, then the available execution can be parallel. We leave exploring the various approaches for future work.

The current implementation only allows the main thread to fork the initial task with one or more induction variables. Furthermore, when the groups of tasks are executing, no other tasks can be forked. Since each thread only has one group of tasks to execute, there
is no distinction between the two ways of scheduling tasks. Removing these limitations in the implementation is left as future work. Once the thread that forked the initial task executes its group of tasks, the thread waits until it is notified that all the other threads finished their work (i.e. a join is performed). Thus, the tasks for all the values of the induction variables are executed together with a small amount of overhead.

We have added five optimizations to the scheduler logic that improve the performance of parallel execution.

First, when the thread that initially forked the task with induction variables is waiting for all the other threads, it should not yield to other threads. We refer to this optimization as main-thread-no-yield. The benefit of yielding is that other threads, such as a compilation thread, can perform work. The disadvantage of yielding is that it reduces the responsiveness of the main thread. If the tasks are relatively small, then performance will most likely be better when the main thread does not yield.

Second, when a loop-based task is forked, and groups of tasks are being generated, the worker threads can be notified that they will soon receive a group of tasks to execute. We refer to this optimization as active-notify. The worker threads will then busy wait by spinning in a loop checking if work has arrived for a short period of time before yielding, hoping that work will arrive soon. The length of this period of time can be changed. Although busy waiting tends to be viewed as a drain of resources because no useful work is performed, we have found that a small amount of busy waiting is good. The reason is that busy waiting allows a thread to execute work as soon as it is given to the thread, without having to wait. If a thread yields, then there is a delay before it will execute again and execute the work. Therefore, a small amount of busy waiting can be beneficial.

Third, the access to the variable that shows whether a thread is finished executing its task or not is performed outside of a synchronized section. We refer to this optimization as no-sync. Accessing the variable without synchronization works correctly because the variable is a volatile boolean variable. Although synchronization would allow more
complex information to be kept track of, and tends to be a natural approach to share information between Java threads, we have found the overhead of synchronization is too large (see Section 8.7.4).

Fourth, for loop-based tasks, tasks that should execute in parallel are made to execute sequentially if the number of induction variable values, which correspond to loop iterations, in a loop is too small. Executing sequentially has the benefit that the overhead of parallel execution is avoided. If the amount of work performed in loop-based tasks is small enough that the execution is performed faster sequentially than in parallel, then the tasks should be executed sequentially. The threshold that is used needs to be chosen carefully because some loops may perform a large amount of work with few iterations, while other loops may perform very little work with a large number of iterations. Regardless, having the option of executing parallel tasks sequentially can be used to avoid slowdowns.

Fifth, for tasks without induction variables, which are recursive, the maximum nesting level, i.e. how deep in a recursion tasks can be sent to different processors, can be changed. A smaller maximum nesting level will reduce overheads, while a larger maximum nesting level will increase the number of tasks available for scheduling, thus allowing processors to be better utilized.

7.6 Hardware Transactional Memory and Java

Several challenges have to be met to execute Java programs on an HTM system. Further challenges arise from executing Java on an HTM simulator. In this section we describe both sets of challenges and the way they are addressed.

Although, HTM systems handle the issue of dealing with dependences, they only work well when dependence violations are irregular and infrequent. Regular or frequent dependence violations lead to problems because they cause transactions to consistently
fail. Such regular or frequent dependence violations usually arise when a memory location is written in various parts of a program. Unfortunately, many typical uses of Java programs tend to have such problematic writes. This section contains a description of when such writes occur and how they can be avoided.

The first source of memory locations that could be written regularly is memory allocation. The memory allocator has centralized data structures to ensure that everything is in a consistent state. Unfortunately, memory is allocated very often in Java programs, especially in many parts of the Java class library, for example, when adding elements to a list. Therefore, both creating new objects and calling many parts of the Java class library need to be outside of transactions\textsuperscript{14}.

The second source of memory locations that could be written is the internal code of the JVM. A JVM performs many operations, including compilation and garbage collection. Since the program has no control over these, they could interrupt the execution of code in a transaction and lead to the transaction failing.

We have evaluated the possibility of using three JVMs: the Jikes RVM, Sun’s JVM, and the GNU Compiler for Java (GCJ). Both the Jikes RVM and Sun’s JVM perform many tasks, including compilation, at runtime. Therefore, they would need to be modified considerably before they can be used on an HTM system. Furthermore, the HTM simulator we use, LogTM [55], is based on a SPARC architecture, which the Jikes RVM cannot be executed on. We have chosen LogTM [55] because of its use by many researchers. The acceptance of many researchers increases our confidence in the system being both accurate and robust. In contrast to the Jikes RVM and Sun’s JVM, GCJ compiles all the Java code ahead of time and generates an executable that contains compiled code. The result is that when GCJ is used the Java programs can execute transactions

\textsuperscript{14}If memory allocation is infrequent it can be escaped, although then it cannot be undone. If memory allocation is frequent, then the memory allocation system must be made transactional as well, which is beyond the scope of this work.
without the JVM causing problems. Therefore, we use GCJ to run Java programs on an HTM system in our LogTM prototype, which is described in the Section 7.7.

The third source of problematic writes is synchronization. The reason is that synchronization works by using a single memory location, a lock, to determine when code can be executed. Furthermore, synchronization is performed as part of the Java Native Interface; whenever a native method is called or returned from, synchronization is performed. Thus, both synchronization within the code and calls to the Java Native Interface (JNI) must be avoided within transactions.

Unfortunately, communication with an HTM system by sending it messages, such as transaction begin and commit messages, has to be performed by native code. The calls to native code must be made directly, without the use of JNI. However, GCJ provides an alternative native interface, called the Compiled Native Interface (CNI), that does not use synchronization. Therefore, we use CNI to access the transactional memory operations.

The fourth source of memory locations that could be written is the program itself. Although transactional memory should deal with this situation, false aliasing causes a problem. The reason is that HTM operates on a cache line granularity, while Java stores data on a word granularity. Therefore, two separate threads can write to two different fields and the HTM can see a dependence violation if the two fields are on the same cache line. We avoid this problem by ensuring that all variables that have such a problem are put on a separate cache line. The variables are separated by putting dummy fields that are never used before and after the variable.

The false aliasing can occur not only in applications, but in the parallelization framework as well. This problem can be avoided by making each variable written by the framework be put on a separate cache line.

For these issues to be properly addressed, a JVM needs to be written from the ground up with HTM support. However, even without such a JVM, HTM can be useful for Java
programs if the JVM does not interfere with program execution, if variables are padded and transactions do not contain memory allocation, synchronization, and JNI calls.

Furthermore, special care has to be taken for the Java programs to execute properly on an HTM simulator. First, conflicts between transactions or between transactions and non-transaction code resulting from frequent writing of the same variable need to be avoided. Thus, the parallelization framework is divided into phases, with each phase having different per thread variables, each of which is only written once. Second, transaction nesting results in conflicts between the nested transactions. Our approach to dividing tasks into transactions uses non-nested transactions. Thus, this problem is avoided. Third, the program needs to be structured in a way that once the transaction system is started, the only task that the main thread can perform is to create worker threads and then wait for them to complete. The worker threads are attached to specific processors and perform all the work. Fourth, the benchmarks are divided to ensure that each benchmark has only one computational component, avoiding any interference between components. Fifth, the transaction system is not able to deal with interfaces and instance of tests. Therefore, inheritance is used instead of interfaces, and instanceof tests are replaced by tests against an identifier which needs to be different for each class in a class hierarchy. When all of these issues are handled, Java programs can execute on an HTM simulator.

### 7.7 LogTM Prototype

Since the Jikes RVM cannot run on the HTM simulator, we have created the LogTM prototype to evaluate the effectiveness of using traces to parallelize programs that run on an HTM system. This prototype is implemented as a set of macros that expand to Java source code that communicates with the HTM system via CNI. The macros need to specify the starts and ends of tasks, the forks and joins, as well as the induction and
reduction variables. The prototype is not fully automatic because these macros need to be inserted manually into the program.

The prototype has a packaging component, a dependence component, and a scheduler. The result of the grouping component in the Jikes prototype, which extends the Jikes RVM, is used to identify the tasks for the LogTM prototype. Indeed, the information generated by all the components of the Jikes prototype has been used to help specify the input to the macros. The packaging component of the LogTM prototype is simpler since it does not have to worry about compiling only one method at a time. Furthermore, dependences do not need to be identified, since the macros specify induction and reduction variables, while all other dependences are handled by HTM. Also, the scheduler only assigns tasks directly to processors, and does not use a work queue. Thus, the scheduler is simpler as well. Therefore, the implementation issues that arise in the Jikes prototype are avoided in the LogTM prototype.

Although some kind of method library would have been easier to create, local variables of methods have to be accessed, which cannot be accomplished by using methods. Another potential approach is to use a bytecode rewriter. However, rewriting the bytecode has many limitations. Further, macros are much easier to use. The macros automate a large portion of the code insertion, and the remaining manual step of inserting macros is relatively simple and mechanical. In the future, when JVMs are more integrated with HTM systems, we expect the code insertion step will be fully automatic.

The prototype needs to be designed in a way that avoids false sharing. Therefore, all variables that can be accessed by more than one thread are per thread variables that are stored in arrays, with indices far enough apart such that each variable is in its own

\footnote{Our initial attempt at a prototype used a bytecode rewriter, and we found four disadvantages. First, there is no efficient way to properly test if a virtual method should be invoked or not. Second, traces cannot contain parts of constructors because constructors must be separate methods. Third, traces cannot contain parts of multiple methods that access private fields in multiple classes. Fourth, the generated bytecodes of the traces will in all likelihood fail validation because they perform tasks that regular Java programs do not perform, such as executing bytecodes from multiple methods in the same context.}
cache line. Each array has extra space at the end and all indices are offset from the beginning to avoid interactions with data before and after the contents of the array. If such precautions are not taken then the first element of each array would have a false alias with the size of the array.

For this prototype, we have created three types of macros: macros that deal with tasks that have corresponding induction and reduction variables, macros that deal with tasks that do not have corresponding induction and reduction variables, and macros that deal with initialization. The macros will ensure that all the tasks are packaged properly, that instructions that handle dependences are inserted, and that the tasks will execute in parallel. The dependences are handled by removing dependences on induction and reduction variables, having joins wait for the corresponding forks, and starting and ending transactions at the appropriate points in the program. These transactions are made to commit in sequential program order. Thus, the program is parallelized and all dependences are handled properly. The following is a brief description of these three types of macros.

7.7.1 Macros for Tasks with Induction and Reduction Variables

To deal with tasks with induction and reduction variables, two macros are needed: a TASK macro and an INSERT_TASK macro. The TASK macro is responsible for defining what the task is. The INSERT_TASK macro is then responsible for outlining that task into a different method.

Each call to the TASK macro specifies information about an induction variable, zero or more reduction variables, and the task’s instructions that operate on a single value of the induction variable and produce a single value per reduction variable. The macro will expand to code that traverses the induction variable and stores its values into an
array accessible by the tasks, assigns different parts of the array to different tasks, forks the tasks, waits for them to complete, and then combines the reduction variables at the end. Furthermore, the macro saves its information to allow the task instructions to be inserted later into the body of a new method. The INSERT_TASK macro takes this saved information from each call to the TASK macro and uses it to create tasks in their own methods. These tasks will deal with the induction and reduction variables, begin transactions, execute the tasks’ instructions, and commit transactions in sequential program order.

A regular macro system expands macros and does not allow them to communicate. Therefore, we have created a special macro processor for the task. The macro processor expands the TASK macros that exist in a file, storing the relevant information, and when an INSERT_TASK macro is encountered, it is expanded by using the stored information.

### 7.7.2 Macros for Tasks without Induction and Reduction Variables

We have also created four macros for tasks without induction and reduction variables. For the current implementation of the prototype, these tasks need to span entire recursive methods and the work phase must occur after all the fork phases. An extension that incorporates outlining and interleaved fork and work phases has been left for future work. The four macros are the following: START_TASK keeps track of the nesting level, sends messages to other threads informing them to start work, and starts transactions; FORK assigns the work that is about to be invoked to a thread and then performs the recursive method invocation; JOIN waits for the forked method to be completed and then reads the resulting data; and END_TASK saves the result of the method.

---

16 The array is divided equally and distributed between each thread in sequence according to the thread identifier, except that the last task is always assigned to the current thread.
These macros indicate the important parts of tasks, and transform the tasks to execute in parallel, with transactions inserted correctly. The START_TASK and END_TASK macros insert instructions that are responsible for starting and stopping transactions based on the nesting level of the task being executed. The START_TASK and FORK macros are responsible for creating the appropriate first and last variables. The JOIN macro is responsible for creating joins that wait for the corresponding forks to complete. The macros generate code that has somewhat complex control flow, which will be illustrated by an example.

Consider execution starting at a FORK macro, which corresponds to a fork. Although a fork usually only notifies a scheduler that a task should execute, the forks generated by the macros operate differently. A fork will execute the task it should fork. The START_TASK of this task is then executed and is responsible for first checking with the scheduler if the task should execute on the current thread or whether it should be sent to a different thread. If the task should execute on the current thread, then it does so and then returns, making sure that the instructions generated by END_TASK are executed. The fork then saves the returned value, and any joins that correspond to the fork do no work. If the task should execute on a different thread, then START_TASK sends the appropriate information to that thread and returns, making sure that the instructions generated by END_TASK are not executed. The other thread will then execute the task. The execution on the other thread will encompass all instructions, including those generated by END_TASK. In the meantime, the first thread is able to continue executing instructions after the fork, until a join occurs that waits for the forked task to complete, and then saves the returned value.

7.7.3 Setup Macros

The final set of macros is used to specify which part of the sequential program needs to be transformed for parallel execution. The transformation can be viewed as putting part
of the \texttt{main()} method inside of one thread and putting the worker thread logic in the other threads. The macros transform the code by putting it inside of a \texttt{run()} method of a Java Thread class, and expanding the method with worker thread logic. Depending on the thread identifier, either the transformed code or the worker thread logic will execute. The \texttt{run()} methods of multiple Thread objects can then be executed in parallel.

Four macros help move the code to a \texttt{run()} method. \texttt{RUN\_START} starts the declaration of the run method and needs to be followed by the initial sequential code; \texttt{RUN\_EXEC} will make the extra threads wait until they are needed and then perform the appropriate tasks; \texttt{RUN\_POST} ends the declaration of the run method; and \texttt{SETUP} is inserted into the original code of the sequential program to create all the appropriate data structures, start the threads, and then wait until all the threads complete. Other simple macros generate internal variables, fields, and methods used to execute programs in parallel. These simple macros could be made part of the macros described in this paragraph\textsuperscript{17}.

\footnote{We have decided to use the C preprocessor macro system wherever possible because of its wide availability and ease of use (we have decided not to use m4 because it is too complex). However, the C preprocessor macro system does not allow the individual manipulation of parameters when there is a variable number of them. Therefore, an arbitrary number of variable names and types cannot be manipulated one variable at a time. This issue is minor because the macros are trivial, if a little tedious, to write.}
Chapter 8

Experimental Evaluation

In this chapter, we experimentally evaluate the effectiveness of trace-based parallelization. Ideally, a single set of experiments would be performed on a JVM that executes on top of a real HTM system. However, that is not feasible because we do not have a real HTM system to use in our experimental evaluation.

Therefore, a two pronged approach is taken. First, the performance and overheads of trace-based parallelization on a simulated HTM system are measured. A variety of benchmarks are used, including benchmarks with dependences between traces that are not based on induction and reduction variables. Second, the performance is measured on a real system. The benchmarks need to be limited to those that contain only dependences on induction and reduction variables. The evaluation examines the overheads of trace collection and the overheads and performance improvements resulting from automatically parallelizing the benchmarks based on traces.

The remainder of the chapter is organized as follows. First, the platforms, benchmarks, methodology, and metrics used in the experimental evaluation are described. Second, the results of three sets of experiments that evaluate trace collection overheads, trace-based parallelization on the HTM simulator, and automatic trace-based parallelization on a real system are shown. Finally, a comparison is made with related work.
8.1 Platforms

We have created two prototype implementations of our framework: one that uses the LogTM HTM system and executes on top of GEMS and Simics in a simulated system, and another that is an extension of the Jikes RVM. The following are the configurations of the platforms that we use.

8.1.1 GEMS/Simics

We have implemented a trace-based parallelization prototype by creating a set of macros and an interface to an HTM system. For this prototype, we use the Wisconsin GEMS toolset 2.1 [54] on top of the Simics 3.0.31 full system simulation platform [52], and the GNU Compiler Collection version 3.4.6 to build SPARC binaries.

The benchmarks are compiled on a SPARC Solaris 10 system for the SPARC V8+a architecture on Simics with the -O3 optimization flag. The C/C++ code is connected to the Java code via the Compiled Native Interface (CNI).

8.1.2 Simulated System

The simulated architecture is a 16 core CMP SPARC system running Solaris 10. The system consists of in-order, single-issue processors each with private 32KB 4-way L1 data and instruction caches, a shared 16MB 8-way L2 cache, a packet-switched tiled interconnect with 8 clusters of 2 processors, and 16GB of memory. L1 response latency is 2 cycles, L2 response latency is 15 cycles, and memory response latency is 450 cycles. The HTM system is a LogTM system with lazy conflict detection and lazy version management[55].

The simulator restricts our choice in that it can only simulate in-order processors. Although this may seem like a restriction, it really is not. The reason is that in-order processors are being used in multicore systems because, given a fixed amount of area
and power, using more in-order processors can lead to better performance than having a smaller number of out-of-order processors.

### 8.1.3 Jikes RVM

We have implemented a trace collection system (TCS) and a trace-based automatic parallelization prototype by extending the Jikes RVM version 2.4.0 [42], which has been modified slightly. The modifications are necessary to decrease the probability of performing garbage collection while the optimizing compiler and prototype are executing. Unfortunately the garbage collection interacts badly with the optimizing compiler and the prototype\(^1\). The modifications consist of removing some string concatenation in the adaptive system, increasing the “metadata” space fourfold to 128MB, and using 1600MB as a starting heap size.

The Jikes RVM has a fast compiler that generates slow code and a slow optimizing compiler that generates fast code. Initially, all methods are compiled by the fast compiler and, as the program executes, some methods are optimized. To determine which methods to optimize, the Jikes RVM samples the running program to determine which method is being executed. We refer to this configuration of the Jikes RVM, with both our trace collection system and trace-based automatic parallelization prototype disabled, as the default configuration. We use the Jikes bytecode compiler version 1.22, the Blackdown JDK version 1.4.2, and the GNU Compiler Collection version 3.3.6 for compilation.

The system used is a Dell PowerEdge 6600 with 2 GB of ECC DDR random access memory (RAM) and four 1.6 GHz Pentium 4 Xeon processors running Debian with Linux kernel 2.6.17. Each processor has an 8 KB L1 data cache and a 256 KB L2 data cache. There is also a shared 1 MB L3 cache.

\(^1\)The purpose of the modifications is to avoid limitations of the Jikes RVM. Certain parts of the optimizing compiler generate data structures that need to be completely generated before they can be correctly garbage collected. Ideally, no memory allocation, and thus no garbage collection, would occur while such data structures are formed. However, some memory allocation may occur, and we try to minimize the occurrences. This limitation needs to be addressed by future versions of the Jikes RVM.
8.2 Benchmarks

To evaluate the effectiveness of our techniques, we use sequential applications from the Java Grande Section 3 benchmark suite [44] and from the JOlden benchmark suite [15, 47]. The benchmarks that are used from the Java Grande benchmark suite are loop-based and the benchmarks from the JOlden benchmark suite are recursive. The benchmarks from Section 3 of the Java Grande benchmark suite that are used are moldyn, montecarlo, and euler. We focused on the loop-based benchmarks in this suite, and the other two benchmarks are recursive. Further, one of the benchmarks requires memory allocation throughout execution, which conflicts with HTM, and the other benchmark uses global data extensively, which prevents parallelization. Thus, these other benchmarks are omitted. The benchmarks from the JOlden benchmark suite that are used are bisort, perimeter, treeadd, and tsp. The other benchmarks in this suite need to perform memory allocation throughout execution. The interactions between the memory allocation and transactions prevent us from executing these benchmarks on the HTM system, and thus these other benchmarks are omitted.

Due to implementation limitations in the Jikes prototype, the prototype is instructed to ignore a number of methods\(^2\). For moldyn, the runiters method is ignored because of a lack of dependence testing. For montecarlo many methods are ignored by the prototype because of a combination of a lack of dependence testing and compilation problems\(^3\). These methods are processResults, initTasks, inc_pathValue, getReturnCompounded, computeMean, computeVariance, computeFluctuationsGaussian, computePathValue, splitString, and readRatesFile. For euler, the calculateDamping and calculateDummyCells methods are ignored because they contain object variables that need to be privatized. Also, the initialise method is ignored because of a lack of

\(^2\)These methods have been found empirically. None of the benchmarks were modified to execute with the Jikes prototype.

\(^3\)The problems are due to optimization passes not being able to deal with the input they are given, either because of bugs in the passes or in our prototype.
dependence testing. For *tsp*, the *conquer*, *makeList*, and *merge* methods are ignored due to limitations of the prototype that create compilation problems. For *perimeter*, the *gtEqualAdjNeighbor* method is ignored because it is a tail recursive method, which the prototype does not deal with effectively. Also, when the optimization level\(^4\) is forced to \(\texttt{O2}\), global code placement is disabled, even though it is usually enabled at \(\texttt{O2}\). The reason is that the code generated by the prototype is not properly handled by the loop invariant code motion phase, which is part of global code placement. Finally, on-stack replacement is disabled due to possible problems it can create with the instruction mapping referred to in Section 7.3.2. The potential problem is that on-stack replacement can add extra instructions in the original method that do not appear in any corresponding new methods that are created. We have not noticed any significant negative performance impact of disabling these two optimizations for the benchmarks used in these results. Furthermore, because the key method in *montecarlo*, *runSerial*, is executed only once, an optimized, and therefore parallelized, version cannot be executed unless the method is only compiled by the optimizing compiler instead of the baseline compiler. Therefore, the Jikes RVM is instructed to initially optimize that method for the configurations that have the prototype enabled. Addressing these implementation limitations is left for future work.

For HTM simulation, the benchmarks are divided into three phases, the *startup phase*, which represents the starting of the JVM\(^5\), the *initialization and memory allocation phase*, which represents the beginning of the program, in which the data structures are allocated and initialized, and the *computational phase*, which represents the computational work performed by the program. Only the computational phase is simulated, due to limitations of the simulator which will be explained in Section 8.4.

\(^4\)The Jikes RVM compiler, like many compilers, has several different levels of optimization that it uses when compiling methods. These are referred to as \(\texttt{O0}\), \(\texttt{O1}\), and \(\texttt{O2}\). \(\texttt{O0}\) has the fewest optimizations enabled and \(\texttt{O2}\) the most.

\(^5\)Quiting the JVM is also included in the startup phase, which could be called the startup and shutdown phase. However, since the shutdown phase is more than ten times shorter than startup, it is omitted to keep the name of the phase short.
To avoid interference between different parts of a computational phase, the *bisort* benchmark is modified to sort the dataset once instead of twice, and *perimeter*, which has two main independent computational phases, is divided into two benchmarks, *perimeter1* and *perimeter2*, each with one computational phase.

### 8.2.1 Input Sizes

Our experimental evaluation is based on using relatively small input sizes. The reason is that we want to evaluate our framework in conditions that allow the framework’s overheads to be seen more readily. Having a relatively small granularity provides such conditions. Further, due to the slowness of the HTM simulator, the input sizes for the LogTM prototype are made smaller than those for the Jikes prototype to keep simulation times reasonable.

It would have been ideal to use different inputs in preliminary and primary executions. However, we have looked at using different input sizes for some of the benchmarks and found that the traces that are produced by trace collection are the same. Therefore, we use the same inputs for the preliminary and primary executions for the Jikes prototype. Since the primary execution of the LogTM prototype uses smaller input sizes, the inputs for this execution are different from those used in the preliminary execution, but nonetheless, the same traces are used.

The input sizes and execution times of the default configuration of the Jikes RVM are shown in Table 8.1. This data is pertinent to the first and third sets of experiments.

The input sizes for the LogTM prototype and the number of cycles simulated for the computational phase in each benchmark without any trace-based parallelization are shown in Table 8.2. The data is pertinent to the second set of experiments.

To further limit the simulation time, the loop-based benchmarks used by the LogTM prototype have been modified. Both *moldyn* and *euler* perform the same computations multiple times, allowing the results to converge on a solution. The two benchmarks have
### Benchmark

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input Size</th>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bisort</td>
<td>2097152 elements</td>
<td>20.03</td>
</tr>
<tr>
<td>perimeter</td>
<td>17 levels</td>
<td>5.55</td>
</tr>
<tr>
<td>treeadd</td>
<td>23 levels</td>
<td>1.826</td>
</tr>
<tr>
<td>tsp</td>
<td>840000 cities</td>
<td>27.04</td>
</tr>
<tr>
<td>moldyn</td>
<td>2048</td>
<td>7.39</td>
</tr>
<tr>
<td>montecarlo</td>
<td>10000</td>
<td>31.76</td>
</tr>
<tr>
<td>euler</td>
<td>256 × 64</td>
<td>28.48</td>
</tr>
</tbody>
</table>

Table 8.1: Input and adaptive configuration execution time.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input</th>
<th>Baseline Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>bisort</td>
<td>16384 elements</td>
<td>27308515</td>
</tr>
<tr>
<td>perimeter</td>
<td>12 levels</td>
<td>46625202</td>
</tr>
<tr>
<td>perimeter2</td>
<td>12 levels</td>
<td>83708071</td>
</tr>
<tr>
<td>treeadd</td>
<td>17 levels</td>
<td>35898088</td>
</tr>
<tr>
<td>tsp</td>
<td>4095 cities</td>
<td>17674471</td>
</tr>
<tr>
<td>moldyn</td>
<td>864</td>
<td>54633152</td>
</tr>
<tr>
<td>montecarlo</td>
<td>32</td>
<td>27896928</td>
</tr>
<tr>
<td>euler</td>
<td>256 × 64</td>
<td>67250194</td>
</tr>
</tbody>
</table>

Table 8.2: Simulation configuration.
been modified to only perform the computations once. Furthermore, the computational phase of `euler` has been modified to only contain a single parallel section, which is part of the `calculateDeltaT()` method. This section is chosen because it is the first such section. The remainder of the computational phase has been removed. Also, `montecarlo` has memory allocation inside of its computational phase. The benchmark has been modified to have its computational phase divided into memory allocation, work, and post processing stages. Only the work stage is parallelized and kept in the computational phase. Also, for `moldyn`, the tasks execute in parallel only when the induction variable indicates that there are at least 128 tasks to execute together\(^6\). The value of 128 is a good balance between allowing execution to be in parallel, and avoiding large overheads due to granularity of parallelism being too small.

### 8.3 Methodology

We perform three sets of experiments. In the first set, we measure the overhead of trace collection. The results demonstrate that trace collection overhead can be large.

In the second set, we measure the performance on an HTM system of benchmarks that have been parallelized using our LogTM prototype\(^7\). This set of experiments examines whether dependences that arise when using traces can be handled efficiently by an HTM system. The results show that dependences in trace-based parallelized programs can be dealt with effectively by appropriate hardware.

\(^6\)Using a threshold of 32 instead of 128 has also been looked at. The resulting performance is slightly worse. Using a larger value has not been looked at, although it would probably not lead to better performance. The reason is that the highest iteration count for an execution of the loop is about a thousand, and using a larger threshold would lead to too much sequential execution.

\(^7\)Although direct scheduling is used for this set of experiments, the performance is comparable to that of using a work queue. The reason is that the number of processors used to execute tasks in parallel is matched to the benchmarks in the experiments (e.g. for a recursive method that calls itself twice, the number of processors used is a power of two). If the Jikes RVM could have been used on LogTM, then we would have used a work queue, which would have allowed the evaluation of performance when a mismatch between the benchmarks and processors exists. We leave using a work queue on an HTM system for future work.
In the third set, we measure the performance of automatic trace-based parallelization in an offline feedback directed system, and thus without trace collection overhead. This set of experiments evaluates automatic trace-based parallelization on a real system, and measures the overheads and speedups. Since the prototype that runs on a real system does not handle all dependences, the benchmarks only have dependences on induction and reduction variables. The results show that automatic trace-based parallelization can be performed efficiently.

Together, the three sets of experiments indicate that traces can be used to automatically parallelize sequential programs and that programs parallelized using traces have better performance on parallel systems than the original sequential versions. Thus, using traces can be a viable approach to automatic parallelization.

8.4 Metrics

We measure the overall improvements in performance of the applications due to our techniques using speedup as our metric. The most general definition of speedup is that the speedup of one approach, $A$, over another approach, $B$, is the ratio of the execution time obtained with $B$ to the execution time obtained with $A$. In the context of parallelism, speedup is defined as the ratio of the best sequential execution time to the execution time of the parallel version of the program. For the Jikes RVM, the execution time is calculated based on the elapsed wallclock time measured by the `time` utility\textsuperscript{8}. All graphs are based on geometric means over 5 executions\textsuperscript{9}. The detailed data is shown in Appendix C. The sequential execution time used for comparison is that of the default configuration of the Jikes RVM, which is the best performing of all the configurations.

\textsuperscript{8}The measurement is accurate to the number of clock ticks per second. The number is set to 100 on the system used. Therefore, the accuracy is 10ms.

\textsuperscript{9}Occasionally compiler generated temporary data leads to a crash in the garbage collector. Executions that crashed are discarded and only data for executions that complete successfully is shown.
on one processor irrespective of whether our techniques are used or not\textsuperscript{10}. Thus, our speedup results are with respect to the best possible sequential performance.

The performance improvements of using traces on an HTM system are also measured using \emph{speedup}. Unfortunately, using the preceding definition of speedup is not possible when using the LogTM prototype to evaluate performance. The reason is that the LogTM simulator does not simulate all parts of a program’s execution, and reports only the number of cycles executed between commands to start and stop measurement on worker threads. This restriction forces us to measure the speedup only for the computational phase of each benchmark, and not for earlier phases. More specifically, for the LogTM prototype, the speedup becomes defined as the ratio of the simulated cycles of a sequential version of the computational phase to the simulated cycles of a parallel version of the computational phase on \( P \) processors. The above definition of speedup is consistent with how the transactional memory research community often reports performance improvement [26, 49, 79]. Nonetheless, it precludes two important components of a benchmark: \emph{startup} and \emph{initialization and memory allocation}.

In our system, the GNU Compiler for Java (GCJ) generated executables load libraries and start the JVM in the startup phase. This phase cannot be executed in parallel and constitutes a sequential fraction of a benchmark that is not measured by the simulator. Further, by default, memory allocation cannot be performed within transactions, including in LogTM, and trying to enable such allocation is an active area of research [39, 73]. Thus, the benchmarks are structured so that all memory allocation is done upfront during initialization, and work is performed afterwards in the computational phase. The initialization and memory allocation phase cannot be executed on transactions and is therefore not performed by worker threads. Thus, the phase is executed sequentially and the time it takes is also not reported by the simulator.

\textsuperscript{10}A comparison is made in Appendix C.
Nonetheless, the above definition of speedup allows us to focus on the gains made by the use of traces and HTM to improve the performance of the benchmarks. This is particularly the case because the use of simulation limits the execution of the benchmarks to the smallest of data sizes, which in turn makes the execution time of a benchmark dominated by startup and initialization. For larger data sizes, the computational phase increases in significance and therefore the impact on the performance of that phase is important to examine.

8.5 Trace Collection

The first set of experiments examines the overhead of trace collection. When enabled, the trace collection system receives information about which instructions are executed when these instructions are in methods that are compiled by the baseline compiler. The trace collection system does not receive information about instructions that are executed in methods that are optimized. The methods that are to be optimized can be chosen in several different ways.

One option is to optimize none of the methods. Then the entire program is executed only using methods that are compiled by the baseline compiler, and the trace collection system knows what happens for the entire execution of the program.

A second option is to keep track of which methods are frequently executed, and then to optimize these methods. The Jikes RVM has an adaptive system that has two ways of keeping track of which methods are frequently executed. The first is to use sampling to identify the frequently executed methods. The methods that are identified are then compiled with increasing optimization levels, starting at O0 and proceeding until O2. The second is to add counters to the baseline compiled code, and when a counter reaches a threshold, to optimize the corresponding method at a specific optimization level. Further, the trace collection system can be used to identify which methods to optimize.
In this set of experiments, the overhead is examined when trace collection occurs when the above mentioned strategies are used to decide which methods to optimize.

In the first experiment, the trace collection system is enabled for the entire execution of each program. That allows traces to be collected based on the entire execution of a program. However, all the methods in each program are only compiled with the baseline compiler. Therefore, the execution is not reflective of how the programs would usually be executed.

Thus, in the second experiment, the optimizing compiler is enabled, and when methods are identified as frequently executed, they are recompiled with the optimizing compiler. When the trace collection system is running, it identifies the top methods of the traces as the frequently executed methods that need to be optimized. A top method of a trace is the earliest method on the call stack out of all the methods on a trace [9]. The resulting execution is similar to the typical scenario where most of the program's execution consists of optimized methods.

There are several tradeoffs between the approaches taken in the two experiments. When methods are optimized, less trace collection occurs and therefore overhead is reduced. However, the overhead relative to the execution time may not be reduced because the optimized part of the program will execute faster. Also, the speed of the trace collection system affects when methods are identified as frequently executed. If the trace collection system is slow, frequently executed methods will be identified later, which leads to less of the execution being on optimized methods. The result is higher overhead. Furthermore, there may be fewer traces collected, because collection does not occur for the entire program. Since the focus of this thesis is parallelization, the study of these tradeoffs is left for future work; only the actual overheads are reported.

Figure 8.1 shows the inverse of the normalized execution time of a number of configurations relative to when the baseline compiler is used by the Jikes RVM. When the baseline compiler is used, the adaptive system is turned off. The y-axis is the inverse of
the normalized execution time relative to using the baseline compiler, and the lower the data point, the slower the configuration\textsuperscript{11}. A number less than 1 indicates a slowdown. The x-axis contains the benchmarks and a geometric mean over the benchmarks, referred to as \textit{geomean}. The data points correspond to only using the baseline compiler, only using the baseline compiler while still having the Jikes RVM insert counters that keep track of what should be optimized, and using a trace collection system\textsuperscript{12}. The threshold for events that start trace recording has been set to 100. Previous work \cite{9} has shown that the threshold has a small effect on trace collection quality and therefore a threshold of 100 is chosen arbitrarily. The data points are referred to as \textit{Baseline}, \textit{Baseline with counters}, and \textit{TCS}, respectively.

![Figure 8.1: Whole program information collection.](image)

The overhead of the trace collection system is usually over 30\%. Therefore, the results indicate that trace collection can be a large source of overhead and needs to be handled properly. We have made early attempts to reduce this trace collection overhead, and

\textsuperscript{11}The y-axis can be viewed as speedup relative to using the baseline compiler, although since most of the values are less than 1, the other configurations are slower.

\textsuperscript{12}The trace collection system is configured to write the traces to a file at the end.
these attempts are described in Appendix B. Since the trace collection system is an offline feedback directed system, trace collection occurs in the preliminary execution and is not part of the primary execution. Thus, the overheads of trace collection do not affect the speedups obtained by trace-based parallelization.

Figure 8.2 shows the performance when runtime information is used to recompile frequently executed methods by the optimizing compiler. In this figure, part of the execution is performed by methods from the benchmark that are optimized. Thus, the execution is different from that presented in Figure 8.1, for which the benchmark methods are only baseline compiled. The significance of Figure 8.2 is that it indicates how well the TCS can perform as an online system. The y-axis is the inverse of the normalized execution time relative to using the Jikes RVM’s default sampling information to determine which methods to optimize. The x-axis contains the benchmarks and a geometric mean over the benchmarks. The data points correspond to the default adaptive configuration, the adaptive configuration which uses counters, a trace collection system set that makes the optimizing compiler optimize methods at $O_1$, and a trace collection system that makes the optimizing compiler optimize methods at $O_2$. The data points are referred to as Default, With counters, TCS $O_1$, and TCS $O_2$ respectively. The Default configuration optimizes frequently executed methods at $O_0$ and if it detects some methods to be executed very frequently then it compiles them at $O_1$, and then at $O_2$. The With counters configuration optimizes methods at $O_1$.

The results show that using a TCS leads to a slowdown. For three benchmarks, bisort, treadd, and moldyn, the performance of TCS $O_1$ is within 10% of the Default execution time. Therefore, although the TCS is slower, as shown in Figure 8.1, the TCS could be effectively used online for these benchmarks. The correct methods to optimize are identified by the TCS. Further, these methods are identified quickly enough for the TCS execution to have only a small negative effect on overall execution time and for any possible delay in their optimization to be negligible. However, for TCS $O_2$ the
performance degrades more, which indicates that the extra time spent in compilation has a negative impact on performance, while the benchmarks do not benefit from the added optimization. The extra time spent in compilation has two negative effects. First, it can delay the optimization of other methods. Second, it increases the amount of time spent compiling methods at runtime.

The *tsp* benchmark is sensitive to the optimization level used to compile a few key methods. Therefore, the TCS can be used effectively as an online system for this benchmark if it can correctly identify which optimization level to use.

For the remaining three benchmarks, the performance of the TCS is more than 10% lower than the *Default* configuration. Further, the performance suffers more than when no optimization is performed. There are three possible sources of this reduced performance. First, the execution of the optimized methods may be sufficiently fast that the overhead of monitoring only part of the execution with the TCS is a larger percentage of the execution than when no methods are optimized. Therefore, the relative size of the overheads may increase. Second, optimization may be delayed because the TCS is slower in deciding
which methods to optimize or the methods are given in a bad order relative to the adaptive system of the Jikes RVM. Third, the TCS may not optimize certain methods that should be optimized. Such a scenario can occur since only the top methods are optimized. These three sources of reduced performance can be overcome by making the TCS efficient, ensuring that good heuristics are used to identify what the traces are, which methods to optimize, and which optimization level to use. Thus, before traces can be used to make optimization decisions, more research needs to be performed. We leave this research for future work.

8.6 Traces and Hardware Transactional Memory

The second set of experiments examines the ability to deal with all types of dependences when performing trace-based parallelization. Four experiments are performed.

The first measures speedup. The experiment shows that the computational phase of the benchmarks benefits from trace-based parallelization on HTM. The second experiment reports on the performance impact of dependences caused by false sharing. The third experiment gives an analysis of the sequential execution of the benchmarks. The analysis is used to reason that it is possible to mitigate the impact of startup as well as initialization and memory allocation on performance. Although the benchmarks, when partitioned into tasks, do not contain inter-task dependences, the interaction with the hardware results in dependences that lead to failed speculation that the HTM deals with by re-executing transactions (as will be seen in Section 8.6.2). Therefore, we believe that our evaluation captures what will happen when problematic dependences in the software do exist.

The fourth experiment looks at synthetic benchmarks that do contain dependences. This last experiment shows that in the presence of dependences, trace-based parallelization can be used effectively in combination with an HTM system. The overhead of
executing traces in parallel and ensuring that the execution is equivalent to the execution of the programs in sequential program order is small enough and the amount of parallelism is sufficient enough to achieve speedups. For all experiments, the minimum nesting level is one and the maximum nesting level is set to have one task at that level per processor (e.g. for 8 processors and a recursive benchmark with a breadth of 2, the maximum nesting level is 3).

### 8.6.1 Speedup of Computational Sections

Figure 8.3 shows the speedup when the computational phase of each benchmark is executed on between one and sixteen processors with one worker thread per processor. The y-axis shows the speedup relative to the computational phase without any parallelization. The x-axis contains the number of processors used for computation. The data points correspond to the benchmarks.

![Figure 8.3: Speedup.](image)

A number of observations can be made. First, trace-based parallelization leads to speedups. In particular, the geometric means of the speedups are 0.93, 1.66, 2.79, 3.66,
and 1.84 for one, two, four, eight, and sixteen processors, with \textit{perimeter1} and \textit{perimeter2} being omitted for two and eight processors processors. Thus, our approach is validated, and trace-based parallelization on an HTM system does lead to performance improvement. Second, the average speedup on one processor is 0.93, which indicates that the overhead of executing the transactions based on traces in sequential program order is small enough to not overwhelm any speedups that can be achieved. Third, the performance of \textit{moldyn} suffers considerably on 8 or more processors. The transactions in \textit{moldyn} tend to be very small and decrease in size when more processors are used. LogTM seems to cope badly with such small transactions. Thus, a balance needs to be found when deciding on the size of transactions. Fourth, performance decreases when 16 processors are used by the worker threads. The reason is that the machine only has 16 processors, and if all of them are used by the worker threads, then there will be contention between one of these worker threads and the main thread\textsuperscript{13}. For these benchmarks, the contention is significant enough to reduce speedup\textsuperscript{14}. Having both a main thread and worker threads is a limitation of LogTM, and would not exist on other systems. Fifth, the speedups are higher than those for the Jikes prototype. Although part of the difference is due to only measuring the computational phase, the speedups indicate that using HTM does not hinder performance. Therefore, as long as the overhead of trace-based parallelization can be kept small, performance improvement can be achieved.

Figure 8.4 contains the distribution of simulated cycles between the different parts of execution. The y-axis shows the simulated cycles per processor. The x-axis contains the benchmarks, with between 1 and 16 processors used for computation. The bars

\textsuperscript{13}Unfortunately, the LogTM simulator requires having the main thread perform no work. Otherwise, one main thread and fifteen worker threads could have been used. If the main thread attempts to perform work, then the simulated program does not execute correctly. Using a power of two for the number of worker threads allows us to use direct scheduling and reduces the number of simulations that need to be performed, thus keeping simulation time more reasonable.

\textsuperscript{14}The interaction with the Solaris scheduler seems to be not very robust. We have observed one instance of an application crashing in the Solaris scheduler code if the worker threads do not yield when waiting for more work.
Chapter 8. Experimental Evaluation

represent cycles spent performing work on transactions that commit successfully, referred to as *good-trans*; cycles spent committing instructions, referred to as *commit*; cycles spent aborting or stalling transactions, referred to as *conflict*; cycles spent executing work on transactions that are aborted, referred to as *bad-trans*; and cycles spent in execution that is outside of transactions, referred to as *non-trans*. For the recursive benchmarks with a breadth of 2, *bisort*, *treeadd*, and *tsp*, there are 1, 3, 7, 15, and 31 transactions for 1, 2, 4, 8, and 16 processors respectively. For the recursive benchmarks with a breadth of 4, *perimeter1* and *perimeter2*, there are 1, 5, and 21 transactions for 1, 4, and 16 processors respectively. For *moldyn* there are 736, 1472, 2944, 5888, and 11776 transactions for 1, 2, 4, 8, and 16 processors respectively. For *montecarlo* and *euler* there are 1, 2, 4, 8, and 16 transactions for 1, 2, 4, 8, and 16 processors respectively.

Figure 8.4: Execution time breakdown.
Chapter 8. Experimental Evaluation

The results show why certain benchmarks have low speedups. The two reasons for the low speedups are transactions aborting due to dependences and a large amount of cycles wasted while not executing transactions.

The results show that only bisort and tsp spend time in bad-trans cycles, which indicate aborted transactions. The presence of aborted transactions indicates that dependences exist in these benchmarks. An examination of debug log data shows that transactions are re-executed due to conflicts on dependent cache lines. For bisort, there are 1, 3, 4, and 6 aborted transactions for 2, 4, 8, and 16 processors, respectively. For tsp, there are 1 and 3 aborted transactions for 8 and 16 processors respectively. Although the existence of dependences is due to hardware interaction instead of the benchmarks themselves, the results do indicate that dependences will lead to reduced speedups. On four processors, for the benchmarks without dependences the average\textsuperscript{15} speedup is 3.23, while bisort has a speedup of 1.14 and tsp has a speedup of 2.83. Section 8.6.2 examines the effect of avoiding the false aliasing by padding.

Further, the results show that moldyn has a large number of non-trans cycles. Since these cycles are not spent executing transactions, in which work is performed, these cycles are wasted and only slow down execution. Therefore, the breakdown of execution pinpoints the sources of poor performance for bisort, tsp, and moldyn, which are the three benchmarks with the lowest speedups.

Table 8.3 shows the cycles on good transactions divided by the number of good transactions for each benchmark and processor count. The table shows that moldyn has on the order of tens of thousands of cycles per transaction and the other benchmarks have on the order of millions of cycles per transaction. Therefore, the results indicate that in a real implementation moldyn will be influenced much more by overheads incurred at the starts and ends of tasks because moldyn has fewer cycles over which the overheads

\textsuperscript{15}The average used is the geometric average or geometric mean. The arithmetic average of the speedups is 3.27.
can be amortized. Further, the other benchmarks may have problems in that they have transations with many cycles, which may lead to large working sets.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>bisort</td>
<td>$2.4 \times 10^7$</td>
<td>$7.2 \times 10^6$</td>
<td>$2.9 \times 10^6$</td>
<td>$1.7 \times 10^6$</td>
<td>$2.0 \times 10^6$</td>
</tr>
<tr>
<td>treeadd</td>
<td>$4.7 \times 10^7$</td>
<td>$1.5 \times 10^7$</td>
<td>$6.5 \times 10^6$</td>
<td>$3.0 \times 10^6$</td>
<td>$7.8 \times 10^6$</td>
</tr>
<tr>
<td>tsp</td>
<td>$1.5 \times 10^7$</td>
<td>$5.1 \times 10^6$</td>
<td>$2.2 \times 10^6$</td>
<td>$1.7 \times 10^6$</td>
<td>$1.4 \times 10^6$</td>
</tr>
<tr>
<td>perimeter1</td>
<td>$5.8 \times 10^7$</td>
<td>$1.1 \times 10^7$</td>
<td>$1.8 \times 10^7$</td>
<td>$4.2 \times 10^6$</td>
<td>$1.8 \times 10^7$</td>
</tr>
<tr>
<td>perimeter2</td>
<td>$9.2 \times 10^7$</td>
<td>$3.1 \times 10^4$</td>
<td>$1.4 \times 10^4$</td>
<td>$7.6 \times 10^3$</td>
<td>$2.7 \times 10^5$</td>
</tr>
<tr>
<td>moldyn</td>
<td>$7.4 \times 10^4$</td>
<td>$1.3 \times 10^7$</td>
<td>$6.6 \times 10^6$</td>
<td>$3.3 \times 10^6$</td>
<td>$6.7 \times 10^6$</td>
</tr>
<tr>
<td>montecarlo</td>
<td>$2.7 \times 10^7$</td>
<td>$3.1 \times 10^7$</td>
<td>$1.6 \times 10^7$</td>
<td>$8.5 \times 10^6$</td>
<td>$7.4 \times 10^6$</td>
</tr>
<tr>
<td>euler</td>
<td>$6.2 \times 10^7$</td>
<td>$9.2 \times 10^4$</td>
<td>$1.4 \times 10^4$</td>
<td>$7.6 \times 10^3$</td>
<td>$2.7 \times 10^5$</td>
</tr>
</tbody>
</table>

Table 8.3: Average cycles per good transaction.

### 8.6.2 Padding

False aliasing can be avoided by padding variables that cause false aliasing and ensuring that they are on their own cache lines. However, the result is an increase in the size of the working set and a loss of locality. Instead of having a single cache miss for multiple variables on the same cache line, multiple cache misses occur. The trade-off is examined in this section.

Figure 8.5 contains the speedups for the benchmarks that have false aliases with and without padding. The y-axis shows the speedup relative to the computational phase without any parallelization. The x-axis contains the number of processors used for computation. The data points correspond to the benchmarks. Each of the two benchmarks with dependences due to false aliasing, `bisort` and `tsp`, has its own variant with padding, `bisort-pad` and `tsp-pad`. The padding is added between each of the fields in the class definition such that each field has padding before and after it. The padding is sufficiently large to ensure each field is on a separate cache line. The y-axis has a smaller scale than that in Figure 8.3. Performance degrades at sixteen processors, which is due to interference from the main thread, an artifact of the simulator.
Although padding removes the false aliases for both benchmarks, padding does lead to worse performance than allowing the false aliasing to exist. The main reason for the poor performance of padding is the increase in cache misses. Figure 8.6 contains the number of cycles taken by memory accesses. The y-axis shows the number of cycles. The x-axis contains the benchmarks. Each of the two benchmarks with dependences due to false aliasing, bisort and tsp, has its own variant with padding, bisort-pad and tsp-pad. The bars represent the number of cycles taken by load hits, store hits, load misses, and store misses on transactions for the L1 cache.

The graph shows that padding leads to a large increase in miss cycles due to cache read misses on transactions. When padding is used, the number of cycles taken by misses increases dramatically. The end result is poor performance. Therefore, adjacent variables that are accessed frequently and tend to be accessed together should not be divided by padding. In such a scenario, the overhead of conflicts within transactions may be lower than the overhead of the additional cache misses incurred by padding.
8.6.3 Sequential Execution Time Breakdown

A breakdown of the sequential execution time of the benchmarks on a real SPARC system is shown in Figure 8.7. The executed benchmarks are single threaded with no transactions. The y-axis is the percentage of total time spent in each phase. The x-axis contains the benchmarks. The total execution times are shown besides the labels of the benchmarks. The effect of using larger input sizes is shown on the right side, and the L denotes that the input size is increased by approximately 32. For recursive data structures, the increase in size has been achieved by adding 5 levels to the data structure, while for non-recursive data structures the structures are just increased in size by 32. The bars represent the startup phase, the initialization and memory allocation phase, and the computational phase. Only the computational phase is put inside of transactions in the experiments.

The input sizes used in simulation are relatively small because of the slowness of the simulator, which leads to the computational phase to be a small fraction of overall execution time and startup time to be significant. When input sizes are increased, the
startup time becomes only a small portion of execution. Thus, startup time will have a minor impact on performance for larger input sizes. However, even with larger input sizes, initialization and memory allocation takes a significant portion of execution for these benchmarks. Initialization and memory allocation not executing in parallel is due to not being able to have memory allocation inside of transactions due to problematic interactions between a JVM and the HTM system, and is not due to using traces. Indeed, in the Jikes prototype, initialization and memory allocation is executed in parallel. In Section 8.7.1 (page 175), Figure 8.22 shows that the amount of sequential execution time spent on tasks that contain groups of traces is in excess of 60% and that other execution, which includes all execution of the instructions of the benchmarks that is not on tasks is less than 20%. In Section 8.7.3 (page 180), Figure 8.24 shows that almost all execution of traces on tasks is parallelized. The only benchmark that has sequential execution of tasks is moldyn, and the amount of execution of sequential tasks is less than 5% of parallel
Chapter 8. Experimental Evaluation

tasks. Therefore, the majority of the instructions of the benchmarks are executed in parallel, including the initialization and memory allocation code\textsuperscript{16}.

8.6.4 Dependences

Two benchmarks have been created to study the effects of dependences on speedup. The first benchmark is a recursive application that contains a method which calls itself with a breadth of 2 to a certain depth, and at each invocation performs some work and, with some probability, writes to common memory locations. Each method invocation corresponds to a task and the work is wrapped in transactions as described in Chapter 6. The second benchmark is a loop-based application that in each loop iteration performs some work and then has some probability of writing to common memory locations. The iterations are divided into tasks and executed on transactions. In this section, we first describe the benchmarks. We then present the percentage of execution with conflicts, the number of cycles taken by the computational phase, the speedups, the distribution of execution, the sequential execution breakdown, and the speedups for different types of configurations.

Three different types of configurations of the benchmarks are considered. First, the writing can be to a single cache line and occur once at the end of the work. Second, the writing can be to a single cache line and occur both at the beginning and end of the work. Third, the writing can be to multiple cache lines and occur both at the beginning and end of the work. These types of configurations are referred to as \textit{rec}, \textit{rec2}, and \textit{rec+} for the recursive benchmark, and as \textit{loop}, \textit{loop2}, and \textit{loop+} for the loop-based benchmark.

For each benchmark, the effect of varying the number of writes to a common memory location is examined. The configuration names are appended with “-w” followed by the number of writes on that configuration. For the recursive benchmark, configurations are

\textsuperscript{16}We have also looked at debug log data from the Jikes prototype and the log data shows that the prototype did parallelize the initialization and memory allocation code.
used that have zero, two, three, four, five, and nine conflicting writes. For the rec type of configuration, the configurations are referred to as rec-w0, rec-w2, rec-w3, rec-w4, rec-w5, and rec-w9 respectively. For the rec2 type of configuration, the configurations are referred to as rec2-w0, rec2-w2, rec2-w3, rec2-w4, rec2-w5, and rec2-w9 respectively. For the rec+ type of configuration, the configurations are referred to as rec+-w0, rec+-w2, rec+-w3, rec+-w4, rec+-w5, and rec+-w9 respectively.

For the loop-based benchmark, configurations are used that have zero, two, three, four, five, and six conflicting writes. The configurations have similar names to those for the recursive benchmark, except that the prefix is changed from rec to loop and the number of writes is adjusted. Thus, for example, for the loop type of configuration, the configurations are referred to as loop-w0, loop-w2, loop-w3, loop-w4, loop-w5, and loop-w6 respectively.

Figure 8.8 contains the tasks of the recursive benchmark. Each task is indicated by a square. At the top level is one task. At the next level two, and so on, until there are 16 tasks at the fifth level. The actual number of transactions depends on the number of processors used for communication. With fewer transactions, some of the writes may occur on the same transaction, even if they are on different tasks. The letters indicate the conflicting writes that are added by each subsequent configuration, and correspond to the letters in the configuration name. The seed of the random number generator is kept the same to ensure the sequence of numbers used to determine which tasks perform writes is the same for all executions. The first writes, indicated by a, are in rec-w2, rec2-w2, and rec+-w2. The next configurations, rec-w3, rec2-w3, and rec+-w3, have writes indicated by a and b. This pattern repeats until e and the “-w9” configurations are reached.

\footnote{The probability of a write occurring is some value which, based on the random number generator in Java, results in some number of writes. The probabilities for the recursive benchmark are 8, 16, 18, 19, and 32 percent, while for the loop-based one they are 8, 16, 18, 24, and 32 percent. These have been kept relatively equal between the two benchmarks, and only 19 has been changed to 24 because that is the lowest probability that would cause 5 writes to occur for the loop benchmark.}
Figure 8.8: Tasks with conflicts for the recursive benchmark.

Figure 8.9 contains the tasks with conflicts for the loop-based benchmark. Each task is indicated by a square. There is only a single level that consists of the loop being partitioned into tasks. The first writes, indicated by $a$, are in $\text{loop-w2}$, $\text{loop2-w2}$, and $\text{loop+-w2}$. The next configurations, $\text{loop-w3}$, $\text{loop2-w3}$, and $\text{loop+-w3}$, have writes indicated by $a$ and $b$. This pattern repeats until $e$ and the “$-w6$” configurations are reached.

Figure 8.9: Tasks with conflicts for the loop benchmark.

Figure 8.10 contains the percentage of execution on transactions that are in conflict for the recursive benchmark. The y-axis is the percentage of conflicting execution. The x-axis is the number of processors used by the worker threads. Since one processor results in one transaction, and a single transaction cannot conflict with itself, only two or more processors are shown. The data points are the different configurations. Each configuration and processor count results in a different percentage. However, each of the different types of configurations will have the same percentage. Therefore, the graph is applicable to $\text{rec}$, $\text{rec2}$, and $\text{rec+}$ types of configurations, even though only $\text{rec}$ is shown.

When the number of processors increases, the number of transactions increases, and the size of the transactions decreases. The reason is that when more transactions exist, with a fixed number of writes, it is more likely that when there are more transactions,
fewer of them will conflict. Since the total number of writes stays the same, when a
transaction is divided into multiple transactions due to a larger number of processors,
only some of the resulting transactions may have conflicting writes. Consider the tasks
in Figure 8.8. When eight processors are used, five of the eight transactions at the third
level have conflicts. These five transactions perform writes $e$, $e$, $a$, $d$, $b$ and $e$, and $a$
and $c$ respectively. When sixteen processors are used, although there are sixteen more
transactions, the number of conflicting transactions goes up only by three. The reason is
that at the third level only two transactions will have conflicting writes and at the fourth
level there are six transactions with conflicting writes. Thus, there is a total of eight
transactions at the third and fourth recursive levels with conflicting writes, compared
to five when eight processors are used. The result is a decrease in the percentage of
transactions that have conflicting writes. Therefore, when the number of processors
increases, the conflict percentage should decrease.

Two observations can be made about the results. First, the percentage of conflicting
transactions does decrease. Therefore, the conflicts should not be a bottleneck to
performance, which should improve with an increase in the number of processors being used. Second, some configurations with different amounts of writes are identical with respect to the percentage of execution on transactions with conflicts for a specific processor count. The reason is that only one write is required per transaction to lead to conflict. Thus, additional writes do not change the conflict percentage, and should not affect performance.

Figure 8.11 contains the percentage of execution on transactions that are in conflict for the loop-based benchmark. The y-axis is the percentage of conflicting execution. The x-axis is the number of processors used by the worker threads. Since one processor results in one transaction, and a single transaction cannot conflict with itself, only two or more processors are shown. The data points are the different configurations. The graph is applicable to loop, loop2, and loop+ types of configurations, even though only loop is shown. The same observations can be made for the loop configurations as for the rec configurations. The reason for 2 processors leading to a percentage conflict of 100% is that with two transactions, the first transaction contains the first 8 tasks and the second transaction contains the last 8 tasks. Since both sets of tasks contain conflicting writes, the two transactions conflict with each other and the percentage of conflicting transactions is 100%.

For completeness, the number of cycles simulated for the computational phase of each benchmark without any changes applied is shown in Table 8.4. The rows represent the different types of configurations while the columns represent the successive number of writes. The cycles represent executions with what we considered the longest running transactions that can be simulated in a reasonable time. The running time on a real system is quite short, a little over 0.5 seconds for the rec and loop benchmarks with no writes and no transactions (see Figure 8.16).

Figure 8.12 contains the speedups obtained for the rec configuration type when the computational phase of each benchmark is executed on between one and sixteen proces-
Figure 8.11: Percentage of execution on transactions with conflicts for the loop benchmark.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>21641822</td>
<td>21642045</td>
<td>21693760</td>
<td>21683489</td>
<td>21682086</td>
</tr>
<tr>
<td>loop2</td>
<td>21627196</td>
<td>21627343</td>
<td>21626286</td>
<td>21626487</td>
<td>21626657</td>
</tr>
<tr>
<td>loop+</td>
<td>21860715</td>
<td>22114404</td>
<td>22189679</td>
<td>22302830</td>
<td>22425398</td>
</tr>
<tr>
<td>rec</td>
<td>19937652</td>
<td>19943835</td>
<td>19926031</td>
<td>19927999</td>
<td>19930127</td>
</tr>
<tr>
<td>rec2</td>
<td>19942823</td>
<td>19919091</td>
<td>19961169</td>
<td>19965898</td>
<td>19964895</td>
</tr>
<tr>
<td>rec+</td>
<td>19821068</td>
<td>20197216</td>
<td>20332191</td>
<td>20446623</td>
<td>20537161</td>
</tr>
</tbody>
</table>

Table 8.4: Simulation configuration for dependence benchmarks.
sors with one worker thread per processor. The y-axis shows the speedup relative to the computational phase without any parallelization. The x-axis contains the number of processors used for computation. The data points correspond to the various configurations.

Several observations can be made. First, the speedups for the configurations with conflicting writes are between 1.33 and 1.86 on four processors and between 1.42 and 2.30 on eight processors. The configurations with the lower percentage of conflicting transactions have higher speedups. The configuration with no dependences has speedups of 3.08 and 2.95 on four and eight processors. Therefore, although dependences do affect performance negatively, speedup can be obtained in the presence of dependences. One possible approach to improve performance is to have multiple worker threads per processor to allow for parallelism to exist at a finer granularity, thus decreasing the percentage of transactions in conflict.

Second, some configurations that have the same percentage of conflicting transactions for a specific number of processors have different speedups. For example, rec2 has a lower speedup than rec3 on 8 processors. Although such behaviour may seem counterintuitive,
the reason is that scheduling affects the performance. We found, by looking at the log
data, that in one configuration one of the transactions is delayed sufficiently to avoid
a conflict which would result in that transaction’s re-execution. Thus, speedup can be
sensitive to small differences in scheduling.

Third, performance degrades for all configurations at 16 processors, and for rec-w0 the
degradation starts at 8 processors. The reason is that the sequential execution commit
order stalls execution. The work in a recursive task being started after both child tasks
complete causes a delay in later transactions. Work is being started after both child
tasks complete to mimic the situation of the work being based on the output of the child
tasks. However, subsequent tasks need to wait for two transactions before the tasks can
commit.

Consider the example shown in Chapter 6 in Figure 6.2 on page 98. The transaction
in Task 6 cannot commit until the transaction for the work in Task 2 commits. However,
if the transaction for the work in Task 2 begins only after the transactions for Tasks 3
and 4 complete, then the transaction in Task 6 can commit only after two transactions
execute sequentially. If each transaction takes 1 time step to complete, then Tasks 3 and
4 would complete after 1 time step; Tasks 2, 6, and 7 would complete after 2 time steps;
Task 5 completes after 3 time steps; and Task 1 completes after 4 time steps. Without
that added delay from Task 2 to Task 6, the execution could be performed in 3 time steps
because Tasks 6 and 7 could complete after 1 time step. The impact is most pronounced
when there are no dependences since dependences can cause delays that would have been
there due to this behaviour.

The results indicate that achieving large speedups for recursive programs is difficult.
Either work performed on a task needs to be independent of the output of the child
tasks, or the guarantee of execution corresponding to sequential program order needs to
be relaxed. For example, for Figure 6.2, Task 6 could wait for Task 4 instead of Task
2. Interestingly, after looking at the log data, we have found that scheduling conflicts
between the worker threads and main thread are not a big factor, and transactions start promptly, even on 16 processors.

Figure 8.13 contains the speedups obtained for the loop configuration type when the computational phase of each benchmark is executed on between one and sixteen processors with one worker thread per processor. The y-axis shows the speedup relative to the computational phase without any parallelization. The x-axis contains the number of processors used for computation. The data points correspond to the configurations. The graph shows the speedup for each configuration type with different numbers of processors.

Two observations can be made. First, the speedups for the configurations with conflicting writes are between 1.30 and 1.89 on four processors, and the highest speedup is 9.47 on 16 processors. Therefore, speedups can be obtained in the presence of dependencies for the loop-based benchmark. However, the best speedups are achieved when dependencies do not exist. Second, the speedups are higher than for the recursive benchmark. The reason is the avoidance of the problem of one transaction having to wait for two transactions to execute sequentially before the transaction can commit. Therefore, the speedups do not degrade for the loop-based benchmark. Thus, there is at least

![Figure 8.13: Loop benchmark speedup.](image-url)
one less issue that needs to be addressed by the parallelization of loop-based programs compared to the parallelization of recursive programs.

Figure 8.14 contains the distribution of execution time for the *rec* type of configuration and Figure 8.15 contains the distribution of execution time for the *loop* type of configuration. The y-axis shows the simulated cycles per processor. The x-axis contains the configurations, with between 1 and 16 processors used for computation. For the *rec* type of configuration, there are 1, 3, 7, 15, and 31 transactions for 1, 2, 4, 8, and 16 processors respectively. For the *loop* type of configuration, there are as many transactions as processors. Table 8.5 contains the number of aborts for different combinations of configurations and numbers of processors. Conflicts that do not lead to aborts are handled by the HTM system by stalling transactions. The bars represent cycles spent performing work on transactions that commit successfully, referred to as *good-trans*; cycles spent committing instructions, referred to as *commit*; cycles spent aborting or stalling transactions, referred to as *conflict*; cycles spent executing work on transactions that are aborted, referred to as *bad-trans*; and cycles spent in execution that is outside of transactions, referred to as *non-trans*. The results indicate that the more conflicts between transactions, the more time is spent in transactions that are aborted.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>rec-w2</em></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><em>rec-w3</em></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><em>rec-w4</em></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><em>rec-w5</em></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td><em>rec-w9</em></td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td><em>loop-w2</em></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><em>loop-w3</em></td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td><em>loop-w4</em></td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><em>loop-w5</em></td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td><em>loop-w6</em></td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 8.5: Number of aborts for different configurations and processor counts.
Figure 8.14: Execution time breakdown for the recursive benchmark.

Figure 8.15: Execution time breakdown for the loop benchmark.
The distribution of the sequential execution time of the \textit{rec} and \textit{loop} configurations on a real SPARC system is shown in Figure 8.16. The executed benchmarks are single threaded with no transactions and no conflicting writes. The y-axis is the percentage spent in each phase. The x-axis contains the benchmarks. The effect of using larger input sizes is shown on the right side, and the L denotes that the input size is increased by approximately 32. The total execution times are shown besides the labels of the benchmarks. For recursive data structures, the increase in size has been achieved by adding 5 levels to the data structure, while for non-recursive data structures the structures are just increased in size by 32. The bars represent the startup phase\footnote{Quitting the JVM is also included, although it is more than ten times shorter than startup.}; the initialization and memory allocation phase, which cannot be put inside of transactions; and the computational phase, which can be put inside of transactions.

![Figure 8.16: Distribution of execution time spent in parallel and non-parallel sections.](image-url)

The small input sizes used in simulation cause the computational phase to be a small fraction of overall execution time and to be much smaller than the startup phase. However, when input sizes are increased, the startup time becomes only a small portion of execution, and the computational phase is the major part of execution.
initialization and memory allocation phase is short enough that it has a minimal impact on execution time. Thus, when startup time is a small percentage of execution, the majority of the execution of these benchmarks can be parallelized.

The speedups for the different types of configurations exhibit similar patterns as those shown previously for the \textit{rec} and \textit{loop} configuration types. Figure 8.17 contains the speedups for \textit{rec2} configuration types. Figure 8.18 contains the speedups for \textit{rec+} configuration types. Figure 8.19 contains the speedups for \textit{loop2} configuration types. Figure 8.20 contains the speedups for \textit{loop+} configuration types. In these four figures, the y-axis shows the speedup relative to the computational phase without any parallelization and the x-axis contains the number of processors used for computation. The data points correspond to the configurations. The speedups are similar to that shown for the \textit{rec} and \textit{loop} configuration types. Therefore, the results show that speedup can be obtained when dependences exist, regardless of the number of memory locations written and the location of the dependences inside of the transactions.

Figure 8.17: Speedup of \textit{rec2} benchmarks.
Figure 8.18: Speedup of \textit{rec+} benchmarks.

Figure 8.19: Speedup of \textit{loop2} benchmarks.
8.7 Automatic Trace-Based Parallelization

The third set of experiments examines the benefit of automatic trace-based parallelization. In this set of experiments, the Jikes prototype is used on a real system. Four experiments are performed to evaluate automatic trace-based parallelization.

First, we measure the overhead of using our infrastructure on one processor. We break down the execution time and normalize it with respect to the default configuration. The results show that our infrastructure introduces minimal overheads.

Second, we measure the speedup of using our infrastructure on up to four processors. The results demonstrate that our approach to automatic parallelization using traces is viable and delivers improvements in performance.

Third, we analyze the execution time of the programs on four processors to determine where time is spent. The results provide insights into the sources of overhead and the limitations to speedup. The results show that the overheads are reasonable, and that further lowering of overhead will require reducing overheads from multiple sources.
Fourth, we measure the effects on speedup of varying parameters used in the prototype. The effects of varying the options for communicating between threads, the maximum nesting level, and the iteration thresholds are examined. The results show that setting the parameters properly is important to achieving a speedup.

### 8.7.1 Sequential Overhead

Figure 8.21 shows the breakdown of the sequential execution time spent in different parts of the Jikes RVM and the benchmarks when using the default configuration. The x-axis contains the benchmarks. The y-axis shows the amount of time spent in different parts of execution, with 1 being 100%. The time is divided into four categories: execution of tasks, parallelization (i.e. the grouping, packaging, and dependence components), optimization (excluding parallelization), and other activities that mainly include execution of instructions not on traces. We refer to these as Task, Parallelization, Optimization, and Other, respectively. The default configuration has only two bars, Optimization and Other, since it does not execute tasks and does not perform parallelization.

The time spent by the optimizing compiler is a small portion of the total execution time. Therefore, a modest increase in compilation time will have a small effect on overall performance. If the benchmarks can be effectively parallelized, then that will affect the majority of the execution, and should lead to performance improvement even when compilation time will increase.

Figure 8.22 shows the breakdown of the sequential execution time spent in different parts of the Jikes RVM and the benchmarks when using our infrastructure. The x-axis contains the benchmarks. The y-axis shows the amount of time spent in different parts of execution, with 1 being 100%. The time is divided into the four categories Task, Parallelization, Optimization, and Other.
Figure 8.21: Time spent in different parts of execution for the default configuration on one processor.

Figure 8.22: Time spent in different parts of execution when using our infrastructure on one processor.
The results indicate that the majority of time is spent executing tasks. Therefore, if the tasks can be executed in parallel with low overhead, then our framework will improve the performance of these benchmarks.

### 8.7.2 Overall Parallel Performance

Figure 8.23 shows the speedup of seven benchmarks, bisort, perimeter, treeadd, tsp, moldyn, montecarlo, and euler, as well as the geometric mean of their speedups, referred to as *geomean*. The y-axis is the normalized speedup relative to the default configuration, the x-axis contains the benchmarks, and the data points are the speedups of automatic trace-based parallelization when executing on between 1 and 4 processors. With respect to the scheduler logic optimizations described in Section 7.5, the main thread does not yield when it is waiting for the worker threads to complete their loop-based tasks, the worker threads are notified that tasks are about to be sent to them, the worker threads loop for 3200 iterations to wait for the tasks before yielding, no synchronization is used to access the variable that indicates if a thread has completed its task, loops need to have at least 50 iterations before the corresponding tasks are executed in parallel, and the maximum nesting level is set to three. We refer to this configuration as the *usual* configuration.

All applications exhibit a speedup as a result of our techniques. The geometric means of the speedups are 0.89, 1.36, 1.61, 1.83 for one, two, three, and four processors respectively. Furthermore, on four processors the speedups vary from 1.17 for moldyn to 2.87 for montecarlo. Thus, the automatic parallelization performed by our prototype leads to speedups, both for benchmarks that are loop-based and recursion-based. Also, the overheads of the prototype are for the most part small. With the exception of tsp and moldyn, the execution on one processor is within 5% of the default configuration. The largest overhead is for moldyn, which has an inner loop parallelized. Since the inner loop is executed many times, the tasks are scheduled and information to allow them to execute...
on worker threads is sent many times, leading to significant overheads. Because of the overheads, at least three processors need to be used for there to be a speedup. Although the overheads are large, an increase of the input set size would result in more work being done in the inner loop, and thus the overhead could be amortized over more instructions.

Two of the benchmarks, moldyn and montecarlo, have hand parallelized versions from the same source that produced the sequential versions [44]. The above results compare favorably to running these hand parallelized versions of moldyn and montecarlo provided by the Java Grande Forum [44] using the default configuration. The hand parallelized version of moldyn is designed to capture parallelism that is more coarse grain. This design leads to high overheads and the hand parallelized version can only achieve a speedup of 1.02 on four processors. In contrast, our techniques achieves a speedup of 1.17. Our framework parallelizes the inner loop of the benchmark. The only dependences that need to be dealt with in this loop are on induction and reduction variables, which can be handled with little overhead. The hand parallelized version performs a large amount of book-keeping that allows parallelization at a coarser granularity, thus incurring large
overheads. In contrast, our framework parallelizes the application with less overhead, which leads to a larger speedup. For montecarlo, the overheads are small and the speedup is 2.85 on four processors, compared to a speedup of 2.87 for our techniques. Thus, for these benchmarks, our approach achieves performance better than or comparable to hand parallelization.

The results presented in this section demonstrate the benefits of using traces for automatic parallelization. The applications have been effectively parallelized using traces without using source code. Furthermore, for two of the loop-based benchmarks, the resulting performance of the benchmarks when our system is used is better than or comparable to the performance of the same applications when hand parallelized by others [44]. The results show that when taking into account overheads of automatically transforming applications, except for dealing with all dependences, which are dealt with by the LogTM prototype examined in Section 8.6, the performance of applications can be improved by performing trace-based parallelization. Thus, the results are promising and indicate that trace-based parallelization is a viable approach to automatic parallelization.

8.7.3 Parallel Overheads

To gain more insight into where time is spent during parallel execution, we divide execution time into eight parts. This division is performed based on what the main thread does. Execution time is separated into four categories as before: Task, Parallelization, Optimization, and Other. The execution of tasks is further subdivided. When a task executes (i.e. the Task category), time is spent in three ways. First, by passing information to and from the task. Second, by the task processing this information or waiting for other tasks (i.e. coordination between tasks). Third, by executing the program’s instructions. The latter two ways can be kept track of based on whether the task is executing sequentially or in parallel with other tasks. Therefore, the Task category is divided into five subcategories: passing information to and from tasks, a sequential task processing
this information or waiting for other tasks, a sequential task executing the program’s instructions, a parallel task processing this information or waiting for other tasks, and a parallel task executing the program’s instructions. These five subcategories are referred to as called **Calling Overhead**, **Sequential Overhead**, **Sequential Work**, **Parallel Overhead**, and **Parallel Work**, respectively.

Thus, the execution is broken down into

1. **Parallelization** - grouping, packaging, and dependence components,

2. **Optimization** - performing optimizations excluding parallelization,

3. **Other** - execution that is not spent in the other seven parts,

4. **Calling Overhead** - task execution spent in calling and returning from all methods that contain tasks,

5. **Sequential Work** - task execution that is sequential and performs useful work,

6. **Sequential Overhead** - task execution that is sequential and is not useful work (i.e. execution that is idle or part of our framework),

7. **Parallel Work** - task execution that is parallel and performs useful work, and

8. **Parallel Overhead** - task execution that is parallel and is not useful work.

Figure 8.24 depicts the breakdown of the execution time spent by the default configuration in different parts of the Jikes RVM and the benchmarks when four processors are used. The y-axis shows the amount of time spent in different parts of execution of the main thread, with 1 being 100%. Therefore, the different parts of execution indicate the activity of the main thread. When multiple threads are executing in parallel, the effect of the other threads can only be seen in that they add parallel overhead to the main thread and prevent the main thread from performing useful parallel work. The x-axis contains the benchmarks. The time is divided into the eight parts described above.
Chapter 8. Experimental Evaluation

The execution is divided relatively evenly, and there is no one single source of overhead that significantly affects the performance of all the benchmarks. However, some sources of overhead impact performance more than others. The major sources of overhead are different for each benchmark. The three benchmarks bisort, treeadd, and moldyn have parallel overheads of more than 15%. The three benchmarks perimeter, treeadd, and euler spend more than 15% of time in other mainly non-task execution. Further, more than 15% of time spent in perimeter, treeadd, moldyn and euler is taken by the optimizing compiler. All other sources of overhead are less than 15% of execution time. Also, the benchmarks with these sources of overhead are the benchmarks that have the lowest speedups. Therefore, to improve performance even more, the three main sources of overhead, task coordination, non-task execution, and compilation overhead, need to be focused on.

Figure 8.24: Time spent in different parts of the main thread when benchmarks execute on four processors.
8.7.4 Importance of Scheduler Parameters

We have also examined the effects of varying the parameters that affect the scheduler, which are described in Section 7.5. In the usual configuration of the Jikes prototype, the main thread does not yield when it is waiting for the worker threads to complete their loop-based tasks, the worker threads are notified that tasks are about to be sent to them, the worker threads loop for 3200 iterations to wait for the tasks before yielding, no synchronization is used to access the variable that indicates if a thread has completed its task, loops need to have at least 50 iterations before the corresponding tasks are executed in parallel, and the maximum nesting level is set to three. In this section, individual behaviours of this configuration are changed, and the effects on performance evaluated.

Figure 8.25 shows the effects on performance of varying how the main and worker threads communicate. The y-axis is the inverse of the normalized execution time relative to the usual configuration of the Jikes prototype\textsuperscript{19}. The x-axis contains the benchmarks and geometric mean. The data points correspond to the configurations that have one of three changes made from the usual configuration.

Each change corresponds to disabling one of the first three optimizations made to the scheduler logic, \textit{main-thread-no-yield}, \textit{active-notify}, and \textit{no-sync}. as described in Section 7.5. These data points are referred to as \textit{main-thread-yield}, \textit{no-active-notify}, and \textit{sync} respectively.

The results indicate that all three optimizations are important, since all three configurations are slower than the usual configuration. Both \textit{main-thread-yield} and \textit{no-active-notify} result in \textit{geomean} being between 0.95 and 0.90 of the usual configuration. Therefore, disabling the two optimizations that improve responsiveness of the threads has a small negative impact. The last optimization, which allows threads to access the variable

\textsuperscript{19}The y-axis can be viewed as the speedup relative to the usual configuration of the Jikes prototype.
Figure 8.25: Effects of varying communication, normalized with respect to the usual configuration.

that indicates whether tasks have finished their work without using a synchronization section is the most important one. The geomean is 0.72 of the usual configuration. Therefore, using a synchronization section\(^{20}\) leads to large overhead that needs to be avoided. Furthermore, moldyn has the worst performance without these optimizations. For moldyn, the inverse of the normalized execution time is less than 0.8, and can drop to below 0.2 when synchronization is used. This behaviour is expected because moldyn has very small loop-based tasks, and therefore any overheads of parallel execution will be proportionally large.

Figure 8.26 shows the effects of changing the minimum number of iterations needed before tasks are executed in parallel (described in Section 7.5). The y-axis is the inverse

\(^{20}\)The appropriate variable that keeps track of when tasks complete is accessed via “synchronized(this) { . . . = x; }” instead of just “. . . = x;.” The Jikes RVM uses thin locks that can be changed to fat locks when necessary. Since the state of the variable changes only when a task completes execution, contention is a small factor in the resulting slowdown. Instead, the overhead is due to the large number of times that the locks are acquired and released. Executing moldyn with a version of the Jikes RVM with lock statistics gathering enabled has shown that there is almost no lock inflation: using synchronization leads to an increase of over 90 million thin lock acquisitions and less than 600 fat lock acquisitions.
Chapter 8. Experimental Evaluation

of the normalized execution time relative to the usual configuration of the prototype. The x-axis contains the benchmarks and geometric mean. The data points correspond to the configurations which have the minimum iteration count changed from 50 to either 0 or 100.

![Figure 8.26: Effects of varying iteration thresholds, normalized with respect to the usual configuration.](image)

The two benchmarks that are most affected by the minimum iteration count are moldyn and euler, which have the inverse of the normalized execution time drop to slightly below 0.7 and 0.6 respectively. Since each data point is a geometric mean of five executions, the data points can be affected by the executions differing from each other. The reason for the degradation in performance for moldyn is that some executions are slower than others\(^{21}\). The faster executions are approximately as fast as those of the usual configuration. However, the slower executions are significantly slower, thus decreasing performance. We believe that the reason for the slower execution is that sometimes the optimization of some key methods is delayed. Thus, the non-optimized versions of

\(^{21}\)The execution times can be found in Appendix C.
the methods, which are slower, execute for a longer period of time, resulting in higher execution time. For *euler*, the performance degradation is due to not executing tasks in parallel. Many of the parallel loops in *euler* have an iteration count of 72. Thus, when the threshold is too large, these loops execute sequentially. The results show that the thresholds need to be chosen carefully.

Figure 8.27 shows the effects on performance of the Jikes prototype of varying the maximum nesting level, which controls which tasks can fork tasks that can execute on different processors. The y-axis is speedup relative to the default configuration of the Jikes RVM. The x-axis contains the benchmarks and geometric mean. The data points consist of using a maximum nesting level of two, three, and four on either three or four processors. The maximum nesting level is indicated by $L$ and the processors by $P$. Note that the usual configuration is referred in this graph as $P=4, L=3$.

![Figure 8.27: Effects of varying the maximum nesting level.](image)

The results indicate that for these benchmarks on a four processor system a maximum nesting level of three provides a good tradeoff between flexibility and overhead. The maximum nesting level needs to be large enough to create more tasks than processors.
Otherwise, the performance can degrade. That is the reason for a maximum nesting level of two being too small. Creating too many tasks causes unnecessary overhead. Thus, a maximum nesting level of four is too large. Therefore, the maximum nesting level needs to be balanced to avoid both large overheads and not having enough tasks to ensure that there is enough execution to keep all the processors busy.

8.8 Comparison of Related Experimental Evaluations

We compare the results obtained in our experimental evaluation with the results that exist in related work, which is examined in more detail in Chapter 9. The related work evaluates performance based on varying architectures, programming languages, input sizes, and definitions of speedups\(^\text{22}\). Therefore, comparisons are difficult to make. However, we still present the results to see how well trace-based parallelization performs.

Several researchers examine the speedup obtained by executing parallel versions of the recursive benchmarks used in our experimental evaluation. These include Marron et al. [53], Carlisle et al. [16], Zhu and Hendren [83], and Quiñones et al. [61]. The Java Grande Forum supplies parallel versions of several benchmarks in the Java Grande Suite [44]. We measured the speedups from these parallel versions in Section 8.7.2. Also, several researchers examine the speedup obtained by executing parallel versions of the loop-based benchmarks used in our experimental evaluation. These researchers include Carpenter et al. [18]; Shirako, Kasahara, and Sarkur [66]; and Carlstrom et al. [17]. Finally, Chen and OluKotun [23] automatically parallelize several of the loop-based benchmarks we use.

The speedup results are shown in Figure 8.28, which contains tables a through d. The tables show speedups for four, eight, sixteen, and thirty two processors respectively. For each table, the speedups are shown in the order listed in the previous paragraph, with

\(^{22}\) Some speedups are for the entire program and some for only the computational phase.
the speedups from our Jikes and LogTM prototypes shown first. All speedups are on four processors, except for the work by Zhu and Hendren and Shirako, Kasahara, and Sarkur. The LogTM results show the \textit{perimeter1} followed by the \textit{perimeter2} results separated by a “/”. The results for Zhu and Hendren show the results for eight processors followed by sixteen processors. The results for Shirako, Kasahara, and Sarkur are for thirty two processors. If the related work presents speedups in graphs, then the speedups shown here are estimated.

<table>
<thead>
<tr>
<th>4 Processors</th>
<th>Jikes</th>
<th>LogTM</th>
<th>Marron</th>
<th>Carlisle</th>
<th>Quiñones</th>
<th>JGF</th>
<th>Carpenter</th>
<th>Carlstrom</th>
<th>Chen</th>
</tr>
</thead>
<tbody>
<tr>
<td>bisort</td>
<td>1.62</td>
<td>1.14</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>perimeter</td>
<td>2.48</td>
<td>3.11/3.54</td>
<td>1.0</td>
<td></td>
<td>1.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>treedadd</td>
<td>1.26</td>
<td>2.92</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tsp</td>
<td>2.24</td>
<td>2.83</td>
<td>3.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>moldyn</td>
<td>1.17</td>
<td>2.46</td>
<td></td>
<td>1.02</td>
<td>2.6</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>montecarlo</td>
<td>2.87</td>
<td>3.73</td>
<td></td>
<td>2.85</td>
<td>5.6</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>euler</td>
<td>1.81</td>
<td>3.85</td>
<td></td>
<td></td>
<td>4.0</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8 Processors</th>
<th>Zhu</th>
<th>16 Processors</th>
<th>Zhu</th>
<th>32 Processors</th>
<th>Shirako</th>
</tr>
</thead>
<tbody>
<tr>
<td>bisort</td>
<td>6</td>
<td>bisort</td>
<td>10</td>
<td>bisort</td>
<td>14.9</td>
</tr>
<tr>
<td>perimeter</td>
<td></td>
<td>perimeter</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>treedadd</td>
<td>6</td>
<td>treedadd</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tsp</td>
<td></td>
<td>tsp</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>moldyn</td>
<td></td>
<td>moldyn</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>montecarlo</td>
<td></td>
<td>montecarlo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>euler</td>
<td></td>
<td>euler</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.28: Comparison of speedups in related work.

Three observations can be made from the results. The first observation is that the speedups tend to vary from one related work to another. For example, Shirako, Kasahara, and Sarkur have trouble parallelizing \textit{euler}, while Carpenter et al. seem to be able to parallelize the benchmark effectively. The second observation is that the related work tends to focus on either loop-based programs or recursive programs. The third observation is that except for \textit{bisort}, which suffers the most from false aliasing in the
LogTM prototype, the speedups obtained by trace-based parallelization tend to be relatively close\(^23\) or in between the speedups obtained in the related work. The observations indicate that although higher speedups could be obtained, trace-based parallelization is relatively effective at obtaining speedups of both loop-based and recursive programs. Thus, using traces allows for the effective capture of parallelism that exists in both loop iterations and recursive methods.

\(^{23}\)The speedups for treeadd and tsp are 3 when rounded to the nearest integer both for our LogTM prototype and the related work executing on 4 processors.
Chapter 9

Related Work

Our work builds upon several different research areas, namely traces, parallelization, automatic parallelization, and speculation. In this chapter, we examine some of the related work in each of these areas.

9.1 Traces

A large amount of research has been performed on using traces. The following examines related work in trace scheduling, path profiling, hardware trace systems, software trace systems, Java trace systems, and traces used for parallelization.

9.1.1 Static Trace Scheduling

Fisher [31] was the first to introduce traces and to use them for instruction scheduling. Instructions are scheduled based on whether they are on the most frequently executed trace that has instructions which are not yet scheduled. When instructions are reordered so as to make another path in the CFG invalid, then extra instructions, referred to as compensation code, are added that make that path valid. This work has been extended by many others including Ellis in his Bulldog compiler [29, 51], Howland et al. in their
Horizon compiler [40], Chang and Hwu [21], Hwu et al. [41], and Lowney et al. [51] at Multiflow.

Lowney et al. [51] have created Very Long Instruction Word (VLIW) processors and the Multiflow compiler. The Multiflow compiler uses trace scheduling and aggressive loop unrolling to achieve a high level of instruction level parallelism (ILP). The scheduler selects what it considers is the unscheduled instruction that will execute most frequently, and grows a trace from this instruction. When certain conditions are met, the trace formation stops, the trace is scheduled, and the process repeats.

Hwu et al. [41] have also examined improving the ILP in VLIW processors by using superblocks, which are regions of code that have no side entrances. Side entrances are eliminated by duplicating any code after the code after the first side entrance and making all code that enters the superblock go to this duplicated code. This duplication avoids additional book-keeping.

Hank, Hwu, and Rau [37] have examined compilation based on regions of code instead of methods. Their approach is similar to trace scheduling because they start at a seed node and expand a region from that node based on which connected nodes are executed most frequently. They find that using regions instead of methods can capture important execution characteristics while avoiding the code explosion of aggressive inlining.

Our work differs from the above body of work because the above body of work focuses on instruction scheduling, trying to increase instruction level parallelism, and improving the performance of software on a single processor. In contrast, our work focuses on coarse-grain parallelism and trying to improve the performance of software on multiple processors.

### 9.1.2 Path Profiling

Traces have also been used to keep track of frequently executed paths of a control flow graph. Ball and Larus [6] collect path statistics using path encodings. They encode paths
to have unique indices, which can be used to access counters associated with the paths. The program is instrumented to maintain an index value that is updated properly when execution goes along a certain path. When the execution of a path ends, the index value is used to increment the counter associated with that path. The resulting information is used only for analysis and not for optimization.

Young [80] also performs profiling based on paths. The potential next paths for a given path are kept track of. At execution, the program uses this information to quickly find what path is being followed at a given instruction by comparing the current instruction against the potential next paths of the path that existed for the previous instruction. The paths are used for static correlated-branch prediction and path-based superblock scheduling.

Ammons and Larus use path profiles to perform data flow analysis and constant propagation [2]. They create a hot path graph (HPG) that contains a subset of the CFG based on frequently executed paths, which are referred to as hot paths. Control flow analyses are performed on both the CFG and HPG. Then a subset of the HPG is identified for which the analysis differs from that of the analysis on the CFG. This subset is then used to perform constant propagation. They show that their approach identifies certain cases in which values are constants only on hot paths and not at other times. They use code duplication to avoid side entrances, and thus compensation code.

Our work differs from the above body of work for two reasons. The first reason is that the above body of work focuses on algorithms to keep track of paths that are based on updating information based on what is executed, while our work focuses on algorithms to keep track of what is executed based on predefined indices. The second reason is that the above body of work focuses on optimization, namely branch prediction, scheduling, and constant propagation, while our work focuses on parallelization.
9.1.3 Hardware Trace Systems

Traces have also been used in hardware systems. Friendly, Patel, and Patt [32] use traces in the fill unit\(^1\) of a processor to perform optimizations. They change the behaviour of the fill unit because the latency of the fill unit is not on a critical path and the instructions given to the fill unit will always be executed. The result is that aggressive optimization can be performed by the fill unit. Our work differs from the above work because our work focuses on using traces to parallelize programs to run on multiple processors, while their work focuses on using traces effectively to perform optimization on a single processor.

Jacobson, Rotenberg, and Smith [43] examine the predictability of traces in hardware and compare using traces to using hardware based multi-branch predictors. They show that predictions based on traces are often better than those given by hardware based multi branch predictors. Furthermore, Rotenberg et al. [64] examine the effects of using a processor that has an architecture based on traces. The processor consists of a dispatch unit that uses traces and processing elements that can each execute one trace at a time. Further, the registers can keep track of variables local to traces. Traces, as well as aggressive value and control prediction are used to increase ILP. Each trace executes independently, and when a misprediction occurs, then the hardware ensures that only the traces that are dependent on this trace are re-executed. The processor keeps the penalty of misprediction low by having an efficient mechanism to undo the actions taken by mispredicted traces.

The goal of the above work is to parallelize at a fine granularity and exclusively use special purpose hardware to deal with dependences. In contrast, the goal of our work is to parallelize at a coarser granularity and to use software as much as possible to handle dependences, while relying on hardware that is generic when necessary. Also, they do not describe their proposed hardware structures in sufficient detail to know whether an

\(^1\)A fill unit increases performance by collecting and reordering instructions before they are executed.
efficient design of these structures can be created. In contrast, we target coarse grain parallelism and rely on generic hardware that supports transactional memory and has its data structures described in more detail.

9.1.4 Software Trace Systems

Traces have also been used in software. Two systems that use traces for optimization are the HP Dynamo [4, 5] and DynamoRIO [13, 14] systems, which are designed for the PA-RISC architecture and the IA-32 architecture respectively.

When a program executes on HP Dynamo, the program’s binary instructions are interpreted while HP Dynamo monitors this execution. By interpreting the instructions, the actual instructions do not need to be modified. Once HP Dynamo detects traces, it stores them in a cache. When the trace is stored in the cache, its instructions are optimized, and the trace may be linked to other traces. The optimization should lead to better performance than that of the original instructions. Subsequently, when the beginning of one of the recorded traces is reached, this trace is executed from the cache, without interpreting this trace’s instructions. Over time, the majority of execution should be on these recorded traces. If the majority of execution is on the traces, the performance is improved. HP Dynamo starts recording traces when a threshold is reached for executing backward taken branches or trace exits. HP Dynamo can improve performance of programs, even when they are optimized with a high level of optimization (e.g. O4).

DynamoRIO is similar to HP Dynamo. Initially, instructions not on traces are executed, and this execution is monitored. Then traces are recorded and optimized. The differences are that DynamoRIO does not directly interpret instructions not on traces and exposes an application programming interface (API) for optimizing the traces. Instead of interpreting instructions, DynamoRIO copies the instructions into a basic block cache and adds an instruction that redirects control flow to DynamoRIO at the end of each basic block. The instructions in these copied blocks are then executed. Redirecting control
Chapter 9. Related Work

flow allows for monitoring the execution, recording traces, and executing recorded traces. The instructions are executed directly to improve performance and avoid the difficulty of emulating the IA32 architecture. DynamoRIO exposes an API to perform optimizations to allow other researchers to explore various approaches to optimizing individual traces. Mojo is a similar system by Chen et al. [25] that uses a similar scheme on Windows 2000.

Our work differs from the above work because our work focuses on parallelization, while the above work focuses on optimizations to improve performance on a single processor. Further, our work focuses on Java, while the above work focuses on native binaries. Thus, our work requires interaction with the existing compilation infrastructure of a JVM. Compiling and optimizing individual traces is not possible. Instead, the JVM needs to be told which methods to compile and what the compilation should do. This interaction adds an extra layer of complexity.

9.1.5 Java Trace Systems

A number of researchers have looked at the characterization of traces in Java programs.

Bruening and Duesterwald [12] have modified a Java interpreter to collect “hot” traces and have compared the size and dynamic code coverage of these traces to frequently executed methods and loops. They also simulate how the traces, methods, and loops could be compiled. They find that both traces and loops have better code coverage than just methods.

Berndl and Hendren [7] have also looked at traces in Java which they have collected after extending SableVM to include a trace cache. They examine various characteristics of the traces, such as trace coverage and completion ratios. They use a branch correlation graph to form traces. Each node represents a branch, and edges exist between branches that are executed one after another. Each edge has a counter that keeps track of how often the branches along the edge are executed together. When a counter reaches a threshold, which corresponds to the branches being highly correlated, a trace is formed.
based on the edge and successor edges which also have a high correlation. The traces are stored in a cache and then executed from the cache when encountered. The graph is updated over time, and the counters decay, to ensure that recent information has more importance. If a branch becomes less correlated, then the existing traces are examined and modified to avoid this branch if necessary. They have found that this trace formation approach leads to traces with good properties.

Furthermore, Yasue et al. have looked at efficiently collecting path information in Java using sampling [78]. They create a hierarchy of nested graphs that represents the different loop nests of a method, where each graph represents a single loop. When loops exist in a loop nest, then the graphs are nested. For example, for a loop nest with an outer and inner loop, the outer loop would be a graph that has a nested graph that represents the inner loop. Efficiency is achieved by profiling only one level of nesting at a time. For the above example, first the graph of the outer loop would be profiled. Once enough information is collected for this graph, the graph of the inner loop is profiled. All the profiles are then combined together. By profiling each graph individually, the overhead of profiling is kept small.

Our work differs from the above work since our work focuses on trace-based parallelization, while the above work focuses on collecting traces and then analyzing and characterizing these traces.

### 9.1.6 Traces used for Parallelization

Recently, using strongly connected components of traces for parallelization has been done by Wang et al. [74]. From these components, they compute slices, which are collections of instructions that are required to produce an output. They have also defined hardware that handles speculative execution and can execute these slices. They simulate this hardware and show speedups. Their work postdates ours and differs from our work mainly in two ways. First, their algorithms are different. We group traces into tasks, and
then package and execute these tasks in parallel. They unroll loops and form slices from the strongly connected components of traces. Second, their hardware requirements are different. Our framework is hardware independent, although to fully handle dependences, our implementation uses HTM. They require special hardware that is tightly integrated with their approach. The hardware needs to handle both speculative execution and the direct execution of slices.

There are two definitions of traces other than the one used in this thesis, and both definitions have been used for automatic parallelization. The first is whole program traces, which contain all instructions executed by a program. Tournavitis et al. [71] use whole program traces to create a control and dependence flow graph which is then used to perform automatic parallelization. The second is related to representing partially commutative strings, with each character representing a different part of a program. A trace is all possible strings that are valid in some language. For example, if a trace has the two strings, \(ab\) and \(ba\), then the two corresponding parts of the program, \(a\) and \(b\), can be executed in parallel. Céerin [19] has used this second definition to identify possible ways of parallelizing a program. This work differs from ours in that it uses a different definition of traces, and requires information regarding the entire execution of a program. In contrast, we use traces that are small parts of the program.

### 9.2 Parallelization

Many researchers have looked at the parallelization of software programs. Marron et al. [53] perform alias analysis and then use this analysis to hand parallelize several Jolden benchmarks. Carlisle et al. [16] use annotations of futures to extend C to allow for the expression of parallelism. The execution model is based on migrating computation to the processor that contains the data being computed on. Zhu and Hendren [83] use a points-to analysis to perform locality analysis on several programs written in a parallel dialect.
of C called EARTH-C. They show results for 8 and 16 processors, using the geometric mean of the speedups for the three different approaches presented: simple, localized, and advanced. Carpenter et al. [18] propose extensions to the Java language that support multidimensional and distributed arrays, thus improving performance when using Java to perform parallel and scientific computing. Shirako, Kasahara, and Sarkur [66] examine the performance of X10 and the effects of multiple extensions for Java. We present the speedups that they obtained by using X10. The system they use is a 16-way POWER6 system, with each chip having two-cores with simultaneous multithreading (SMT) turned off.

The above work differs from our work because our work focuses on automatic parallelization while the above work does not. Further, our work uses traces as the unit of parallel execution while the above work does not.

9.3 Automatic Parallelization

A large amount of work has been done on traditional automatic parallelization [3]. This work focuses on dependence analysis and parallelization of loops that access arrays, where the analysis is done at compile time and requires source code. In contrast, we focus on traces at runtime and do not require source code.

Work has also been done on parallelization of loops at runtime. For example, Leung and Zahorjan identify parallelism at runtime by using inspector loops [48]. These inspector loops are used to identify a schedule that would allow all the loop iterations to execute in parallel. Also, Rauchwerger and Padua [62] use a similar approach to identify which loops are completely parallel through the use of shadow arrays. They execute the loops in parallel and then test if the parallel execution was valid by keeping three shadow arrays. Each time an array element is accessed, the corresponding elements in the three shadow arrays are modified. The shadow arrays can then be checked to see if
the parallel execution is valid or not. They also perform tests to check for privatization and reduction.

Rus, Pennings, and Rauchwerger [65] use sensitivity analysis to automatically parallelize programs that may have dependences sensitive to input. They statically analyze loop-based programs. The analysis first produces a dependence set, that if empty, indicates that a loop nest is parallel. They then produce a set of predicates that can be tested at runtime, and if found to be true, show that the dependence set is empty, and the loop nest can be parallelized. Although this work performs analysis at runtime, it still focuses on loop iterations and requires source code. In contrast, our work uses traces as the unit of parallelism and does not require source code.

Other researchers have focused on task-level parallelism. Chan and Abdelrahman [20] explore executing Java methods in parallel when these can be shown to be parallel at runtime using symbolic access paths. Johnson, Eigenmann, and Vijaykumar [46] use a min-cut algorithm to divide programs so that a large amount of parallelism exists and there is little overhead. The algorithm is performed offline and can take up to several minutes to compute the thread shapes. Obata, Ishizaka, and Kasahara [58] describe a compiler that subdivides a program into tasks at the granularity of basic blocks, loops, and methods. These can then be scheduled to execute in parallel using static scheduling or dynamic scheduling that executes sections only once their dependence conditions have been met. This work does not use traces as the unit of parallelism and requires source code. In contrast, our work uses traces as the unit of parallelism and does not require source code.
9.4 Speculation

There is a large body of work that deals with speculative parallel execution. We examine related work in the areas of hardware support for speculative parallelization, Java thread-level speculation, and Java on hardware transactional memory.

9.4.1 Hardware Support for Speculative Parallelization

Sohi, Breach, and Vijaykumar [67] increase instruction level parallelism by dividing programs into tasks to speculatively execute on multiple function units. The hardware maintains a single shared register file and disambiguates memory addresses at runtime, only stalling when true dependences exist. Further, the hardware has the ability to undo incorrect speculative execution.

Tsai and Yew [72] use a processor and system on chip hybrid to speculatively execute loop nests partitioned based on compile time heuristics. Instead of keeping track of dependences and execution for each instruction, they keep track of entire threads, where each thread execution is divided into several stages: a continuation stage that computes induction variables and forks a successor, a target-store-address-generation stage that identifies and saves data that may be used by subsequent threads, the computation stage, and a write-back stage. The amount of hardware book-keeping is reduced by keeping track of information per thread instead of per instruction.

Agarwal et al. [1] use postdominance information to partition tasks on their PolyFlow architecture, which is similar to a simultaneously multithreaded system. In particular, they use immediate postdominators of conditional branches as task starting points. The PolyFlow architecture has hardware to keep track of dependences between all instructions that execute in parallel. A divert queue is used to temporarily store instructions that may have a dependence on an instruction that is not yet executed. The hardware has
support for squashing tasks that violate data dependences and for recovering from branch mispredictions.

Steffan and Mowry [69] have examined the potential of thread-level data speculation, which can be achieved by dividing execution of a program into epochs, which are executed speculatively. If there is a data violation then an epoch will be squashed and re-executed. They choose which code regions should be parallel based on manually going through programs and profiler results. They also discuss what features the hardware needs to implement thread-level data speculation, in particular focusing on how the cache hierarchy can be used to maintain speculative information. This work is extended by Steffan et al. [70]. They have proposed hardware that allows thread level speculation. They allow for speculation by an extension of a writeback invalidation-based cache coherence protocol. This allows them to execute what they refer to as epochs speculatively in parallel.

Chen and Olukotun [22, 23] propose hardware to automatically decompose sequential programs into parallel ones. They use the Hydra single-chip multiprocessor with thread-level speculation (TLS) to speculatively execute loop iterations and have created a hardware system to keep track of dependencies at runtime. Their system identifies at which levels loops should be executed speculatively in parallel. They have reported that their analysis would slow down program execution by a factor of over 100 if they had to use software instead of hardware to identify which loops to execute in parallel.

Zhai et al. [81] have looked at scheduling instructions that are to be executed speculatively at compile time to reduce the overhead of transferring scalar values between threads executed in parallel. They have explicit wait and signal operations for each value, and perform dataflow analyses that identify the latest point at which the wait operation can be performed and the earliest that the signal operation can be performed. The dataflow information is used to perform scheduling. Further, they examine scheduling
the instructions past data and control dependences, which requires changing the dataflow analyses and adding compensation code.

Chen et al. [24] propose to use probabilistic points-to analysis to partition programs. The analysis can be used to calculate the amount of dependence that may exist between loop iterations. If the amount is too high, then execution would be sequential, while if the amount is low, then the loop iterations could be executed in parallel. They also use an architecture that keeps track of threads, which are divided into several stages: a continuation stage that computes induction variables and forks a successor, a target-store-address-generation stage that identifies and saves data that may be used by subsequent threads, the computation stage, and a write-back stage.

Du et al. [28] use a cost model based on dependence information to partition loops. They examine how loops can be divided into non-parallel and parallel portions. They use a two processor system with hardware support for speculation. Each loop has only the latter part speculatively executed in parallel, while the first part is executed serially. They combine this with several optimizations including loop unrolling, and value prediction to achieve performance improvement.

Quiñones et al. [61] propose to predict live-in values and to use pre-computation slices to partition programs. They identify which part of a program to execute speculatively by specifying two points, a spawn point and a control quasi-independent point. The spawn point specifies when the speculative execution should start, and the control quasi-independent point specifies the first instruction that should be speculatively executed. From the spawn point to the control quasi-independent point is non-speculative code that will execute at the same time as the speculative code. Further, the speculative code has an initial phase in which the pre-compute slice is executed. The pre-compute slice consists of parts of the non-speculative code that may compute data needed by the speculative code. If the speculative execution is not valid, it is discarded. They use edge counter profiles to find out how frequently different parts of code are executed.
and compare all pairs of basic blocks in each routine to find good spawn and control quasi-independent points.

Liu et al. [50] use heuristics that choose loops, subroutines, and continuations to partition programs. They then try to hoist the spawn points of the task as early as possible. Afterwards, profile information is used to identify which parts of a program should be executed speculatively. This information consists of the amount of overlap that exists between speculative and non-speculative code, the avoidance of cache misses due to prefetching in speculative execution, code size, hoisting distance, and number of mis-speculated executions.

Johnson, Eigenmann, and Vijaykumar [45] examine using heuristics to automatically partition loops. Their compiler instruments the code for a profile run to search through the potential candidate partitions and identify the best one. The candidate partitions are based on assigning one of three options to each function and loop. The first option is to have a function or loop execute a set of threads. The second is to have a function or loop execute on the same thread as the functions or loops before and after. The third option is to have a function or loop execute on the same thread as the function or loop before it and start a new thread for the next function or loop. They use a two stage search that starts with the third option for all functions or loops. In the first pass some of the functions or loops are changed to the second option, and in the second pass some of them are changed to the first option. This search is done in the profile run, to have precise execution time information. The best combination is chosen as the final partition.

Wu, Kejariwal, and Cascaval [77] develop a cost model and use profile information to partition a program. They collect profile information on dependence probability and the distance between dependent iterations of loops to calculate the probability of successful speculation. They use this probability along with the amount of overheads and the sizes of partitions to determine whether speculation is profitable.
Chapter 9. Related Work

The work closest to ours in regards to enabling commits to occur in sequential program order is by Renau et al. [63] and Hammond, Willey, and Olukotun [36], in which hardware allows tasks to be spawned speculatively and a linked list of tasks is maintained to allow an arbitrary number of tasks to commit in sequential program order on TLS systems.

Renau et al. [63] propose hardware that allows for efficient speculative out-of-order spawning of tasks. The main hardware additions are splitting timestamp interval hardware and immediate successor list hardware. The splitting timestamp interval consists of keeping track of timestamps based on the parent task and the location of the spawn point in the parent task, while the immediate successor list keeps track of the sequence in which tasks need to commit.

Hammond, Willey, and Olukotun [36] propose the Hydra chip multiprocessor that supports data speculation. To keep track of speculative data accesses, they add extra bits to the L1 caches and insert secondary cache buffers before the L2 caches. They maintain a linked list of register passing buffers. The linked list is used to identify the order of tasks, and each element stores information about the register values that each speculative task should begin executing with.

In contrast to the above two approaches, our approach, besides using traces as units of parallel execution, does not need to keep an explicit linked list, spawns tasks non-speculatively, and requires interaction with hardware that supports speculative execution instead of designing this hardware. Not using an explicit linked list allows the data structures to be used even if the order in which transactions need to commit is changed from sequential order. The approaches based on an explicit linked list, especially when they are implemented in hardware, cannot be easily modified to change the order in which speculative execution commits.

Our work differs from the above body of work in three ways. First, the above body of work deals with specific hardware platforms to achieve speculative parallel execution. In contrast, our work is not tightly integrated to a specific hardware platform. Rather,
we only require an HTM system with some generic features. Second, our work focuses on parallelization of programs that execute in a virtual machine, while the above body of work focuses on parallelization of programs that execute directly on the hardware. The virtual machine adds extra constraints on the transformation of programs. Third, we use traces as the units of parallel execution, which the above body of work does not.

### 9.4.2 Java Thread-Level Speculation

Pickett and Verbrugge have extended a JVM with software thread level speculation support [59]. They introduce new speculative fork and speculative join bytecodes, and modify a JVM, SableVM, to allow it to execute Java programs speculatively. They have found that many different types of Java bytecodes may interact with speculation, and they describe their approach to address the issue. Their work differs from ours in that we use traces to automatically parallelize programs, while they focus on enabling software speculation inside of a JVM.

### 9.4.3 Java on Hardware Transactional Memory

Carlstrom et al. [17] use their Transactional Coherence and Consistency (TCC) architecture to execute parallel Java programs by using hardware transactional memory. They use continues transactions, where all execution on each thread is on transactions, and the boundary between transactions is indicated by a commit operation. When a transaction commits, its write set is sent to all the processors, which then check to see if a dependence violation has occurred and some transaction needs to be re-executed. Dice et al. [26] examine the potential performance of a best-effort HTM system. They create a hybrid transactional memory model that goes between having hardware and software transactions. This model is specifically designed to work with the best-effort HTM system. Both Carlstrom et al. [17] and by Dice et al. [26] enhance Java programs with the ability to interact with HTM and measure the results. However, the Java programs are already
parallel, and therefore have their parallel regions delineated manually. In contrast, we examine sequential programs and choose regions based on an algorithm implemented in software.
Chapter 10

Conclusion

In this thesis, we have presented a novel approach to automatic parallelization that is based on traces. Our approach extends traditional automatic parallelization by incorporating runtime information in the form of traces, and uses hardware transactional memory to ensure that execution is correct in the presence of dependences. The successful use of traces to perform automatic parallelization can allow a wide variety of programs to be parallelized and take advantage of multiple processors. Thus, the effective utilization of multiple processors by software programs can be achieved without access to the entire source code of the programs, without worrying about the program having certain programming constructs, and without requiring software developers to adopt new programming paradigms.

10.1 Contributions

We have made a number of contributions in this thesis. We have proposed to use traces to perform automatic coarse-grain parallelization. We have proposed a parallel trace execution model; we have created a trace-based parallelization framework; we have proposed an approach to deal with all dependences based on hardware transactional memory; we have implemented two prototypes that realize our framework; and we have experimen-
tally evaluated the viability of trace based parallelization, demonstrating the feasibility of our trace-based parallelization approach.

We have proposed a parallel trace execution model that allows sequential programs to execute in parallel. Parallel execution is achieved by the identification of traces by a trace collection system that monitors the execution of a program, followed by the transformation of the program to allow the traces to execute in parallel.

We have created a parallelization framework that overcomes the challenges of using traces for parallelization and realizes this execution model in conjunction with a trace collection system. The framework is effective both for recursive and loop-based programs. The framework is divided into four components: the grouping component, the packaging component, the dependence component, and the scheduler.

The grouping component creates tasks from traces, which allows for the reduction of overhead and the easier identification of successor traces. The packaging component creates units of execution based on tasks, connects them properly to the rest of the program, and ensures that local variables and method parameters used as inputs and outputs are passed correctly. The dependence component identifies and removes dependences on reduction and induction variables. In addition, hardware transactional memory needs to be used to deal with dependences that cannot be removed. The scheduler assigns tasks to processors. The four components enable groups of traces on tasks to execute in parallel on multiple processors.

To ensure that all dependences are handled properly, we have proposed an approach for hardware transactional memory to commit transactions in sequential program order. Furthermore, we have described how a parallel transactional memory program can be created based on traces and tasks. The combination of the framework and hardware transactional memory allows for the trace-based parallelization of arbitrary sequential programs.
We have also implemented two prototypes, referred to as the Jikes and LogTM prototypes, that realize our framework and allow Java programs to be parallelized based on traces. We have chosen to use the Jikes RVM because it is the most suitable JVM to extend for research purposes, and we have chosen to use LogTM because of its wide use by researchers. Unfortunately, the Jikes RVM cannot be used on the SPARC architecture, which LogTM extends. Therefore, two prototypes are needed. Both prototypes act as offline feedback directed systems.

The Jikes prototype, an extension of the Jikes RVM, automatically parallelizes programs based on traces. We have also created a trace collection system as an extension to the Jikes RVM. The LogTM prototype parallelizes programs based on traces and uses hardware transactional memory to deal with dependences. We have had to identify and overcome a number of issues that arise from using Java on a hardware transactional memory system.

We have demonstrated the feasibility of our trace-based parallelization approach by performing an experimental evaluation on a number of recursive and loop-based Java programs. The evaluation consists of three sets of experiments. First, the performance of the trace collection system is measured. The results indicate that trace collection can be a large source of overhead and may delay optimization. Therefore, an online system that performs trace-based parallelization requires an efficient trace collection system that effectively identifies what to optimize. Second, the performance of the LogTM prototype is measured. The prototype combines trace-based parallelization with hardware transactional memory. The combination allows for programs to be parallelized and all possible dependences to be handled. The speedup of the computational phase that can be obtained by using trace-based parallelization is approximately 2.79 on four processors. Furthermore, experiments that examine the effects of varying the dependences that exist on a pair of simple benchmarks indicate that speedups can be obtained in the presence of dependences. Third, the performance of the Jikes prototype is measured. The results
show the potential performance that can be achieved, and do not take into account trace collection and a mechanism to deal with all possible dependences. The results indicate that automatic parallelization based on traces can both be done efficiently and lead to good performance. The geometric mean of the speedups on four processors is 1.83, and the best speedup is 2.87. For two programs that have hand parallelized versions, the achieved performance compares favorably to that of the hand parallelized versions.

10.2 Lessons Learned

We have learned a number of lessons regarding trace-based automatic parallelization by performing the research in this thesis.

First, trace-based parallelization can be used to effectively parallelize a variety of recursive and loop-based Java programs. Our experimental evaluation shows that speedups can be obtained both on a real system in the absence of dependence handling and on a simulated HTM system that handles dependences.

Second, HTM can remove the burden of identifying all dependences from the compiler. The compiler can analyze and deal with dependences based on induction and reduction variables, and potentially other easy to identify dependences (e.g. linear accesses to arrays). However, the compiler can leave more complex dependences to be dealt with by the HTM system. Thus, execution will be correct, even without the compiler being able to identify all dependences.

Third, using HTM for automatic parallelization is non-trivial, for two reasons. The first reason is that HTM needs more time to become robust. There are many problematic interactions between transactions and existing software, especially with synchronization and memory allocation. These interactions need to be dealt with, especially if synchronization is used to explicitly deal with simple to identify dependences. Also, there are many different approaches to HTM, and which approaches will become standard has not
been determined since few commercial processors have HTM. Therefore, it is not clear which features HTM systems will have and the inclusion or exclusion of some features may be problematic.

For example, our current implementation relies on performing loads and stores that can escape a transaction, and the lack of such a mechanism would be problematic. The second reason is that obtaining speedups for all types of programs may be difficult. HTM execution is inherently parallel, and extra work needs to be performed to ensure that the execution corresponds to sequential program order execution. The approach taken may prevent significant speedups to be obtained, especially for recursive programs. Thus, the use of HTM requires significant effort to work with traces and Java.

Fourth, to help automatic parallelization research, JVMs need to be designed to allow generating arbitrary computation units. The optimizing compiler in the Jikes RVM is designed to use methods that exist in bytecode as the units of compilation. The consequences are that the compiler is not designed to create other types of methods, there is no direct communication between compiling different methods, and methods cannot be divided arbitrarily. A significant amount of effort has been required to overcome these challenges. Therefore, the design of JVMs is important to ease the difficulty of the automatic parallelization of Java programs.

Fifth, grouping traces is essential. Since each unit of execution needs to be designed to execute on any execution context, there is a large cost of execution transitioning from one unit of execution to another. Therefore, increasing the amount of code executed between transitions by grouping traces decreases overhead. Furthermore, predicting control flow of multiple traces ahead of time is difficult. By grouping traces into tasks in an appropriate way, the control flow of the resulting groups of traces can be made more predictable.
10.3 Future Work

Our research provides an approach that can be used to automatically parallelize both recursive and loop-based programs. This approach can be used as the foundation of a large amount of future work. The following are some of the main areas of future work that we have identified.

One area of future work is the exploration of different trace collection systems. In particular, a trace collection system should be efficient and generate traces that can be used to extract a large amount of parallelism. Creating such a system would require efficiently identifying when to start traces, recording traces, identifying when to stop traces, and being able to access the recorded traces. We have performed some initial research on creating an efficient trace collection system. This research is presented in Appendix B.

A second area of future work is the implementation of the framework in an online feedback directed system, which requires properly defining how optimization and parallelization is performed given partial information. The full set of traces of a program will not be known in an online system. Therefore, tradeoffs need to be made between optimizing earlier with less information and optimizing later with more information. If less information is available, then the resulting code may not perform very well. However, if optimization is delayed, then the resulting code is executed for a shorter period of time, thus reducing the impact on overall performance. The tradeoffs need to be carefully studied, and good heuristics need to be found. Furthermore, the execution patterns may change over the execution, which also needs to be taken into account. The benefit of an online system is that the generated traces and the parallel execution of these traces occur for the same execution of a program. Therefore, the potential problem of the execution pattern changing between two executions of the program is removed.

A third area of future work is to implement the framework on an existing HTM system, such as the Azul compute appliance [34]. A potential disparity may exist between
performance on a simulator and on a real system. By executing on existing hardware, such potential disparities are eliminated. Furthermore, the issue of all operations being allowed on a transaction needs to be addressed. In particular, memory allocation on transactions needs to be allowed.

A fourth area of future work is to implement the framework to parallelize programs written in languages other than Java. Although the prototypes target Java programs, our approach is applicable to programs written in other languages as well. Therefore, allowing programs that are written in other languages to take advantage of our approach would be beneficial. Our framework could be realized by extending other virtual machines, such as a virtual machine for Microsoft’s .NET, or by extending binary rewriting systems such as Pin [60].

A fifth area of future work is to increase the robustness of the prototypes, which will require the expansion of the benchmarks used and dealing with all potential interactions between trace-based parallelization and the other parts of the JVM. Expanding the benchmarks will also allow for a more extensive experimental evaluation that can be used to further improve the trace-based parallelization framework. The parts of the JVM that may have potential interactions include on-stack replacement, garbage collection, reflection, the native interface, and exception handling. Furthermore, all possible corner cases that are in the Java specification need to be handled correctly.

A sixth area of future work is to improve the performance of trace-based parallelization. Although there are many areas where performance can be improved, three examples are given: the scheduler, behaviour towards dependences, grouping, and dynamically adjusting parameters.

The current scheduler is relatively simple, and does not take into account many factors that may impact performance. These factors include load balancing, differentiating between different granularities of parallelism (e.g. inner versus outer loops in a loop nest), or information about potential or existing dependences between tasks. Taking into
account more factors will lead to better performance for a wider variety of programs and will allow the parallel execution to scale well to a larger number of processors.

The behaviour of the system in the presence of dependences that limit trace-level parallelism could also be changed. For example, if at runtime there are patterns of certain tasks having dependences between them, then maybe those tasks should be merged or executed sequentially. Alternatively, maybe the dependence is on a value that tends to not change. Then the dependence could be removed, and only reinstated if the value changes.

Also, the way that grouping is performed could be changed. There are many potential grouping heuristics that could be used. Furthermore, if an online system is used, then groups could be changed by using information obtained from previously generated groups.

Finally, many of the thresholds and configuration options that have a set value could be made to change dynamically. The performance effects of different values could be measured for brief periods of time, and then the best values chosen to be used for longer periods of time. The effect would to be find the best possible combination of values for each benchmark, thus improving performance.

A seventh area of future work is to further parallelize programs that are already multithreaded. The benefit is that even more parallelism may be extracted from these programs, improving performance further. The most important change is to allow the scheduler to deal with tasks that originate from multiple threads. Therefore, the scheduler will need to have a policy for resource sharing between the tasks that originate from different underlying threads. Furthermore, the trace collection system may be able to record traces that span multiple threads. That would open up the possibilities of performing various optimizations that take into account relationships between multiple threads.

An eighth area of future work is to explore using alternatives to HTM for dealing with dependences. For example, a runtime test based on dependence information could
be used to find out if dependences will hinder parallel execution. Before a task is to execute, the test could check if the task potentially performs accesses that are in conflict with other tasks. In that case, the execution of the tasks would be serialized. Otherwise the execution could be parallel.
Appendix A

The Potential of Trace-Level Parallelism

A.1 Simulation Methodology and Timing

In this section, details of the simulation of the benchmarks for the feasibility study in Chapter 3 [11] are provided. More specifically, the number of bytecodes executed, the overall simulation time, and how the time is divided is presented.

Jupiter is executed four times for each benchmark. The first three times are for simulation. Each execution corresponds to simulating a different window size. The fourth time is for measuring the time spent by Jupiter and RedSpot without using PIE. These runs are referred to as “Window Size=128”, “Window Size=1024”, “Window Size=8192”, and “No PIE” respectively.

For each window size, 24 configurations are simulated by calling MCP multiple times. The configurations have combinations of either 2, 4, or 8 processors and eight communication costs and types. The eight costs and types are instruction–instruction, trace–instruction, instruction–trace, and trace–trace communication with fixed and variable costs of 0 cycles and trace–trace communication with 20 fixed and 0 variable cost, 0 fixed...
and 20 variable cost, 20 fixed and 20 variable cost, and 80 fixed and 20 variable cost. Twenty four configurations are simulated at once to amortize the overhead of collecting all the trace and dependence information over a large number of configurations.

The execution time is divided into four components. The time spent by Jupiter and RedSpot, the time spent saving data for PIE to use, the time spent by PIE, and the time spent by MCP. These four different parts of simulation are referred to as $Jup/RS$, $State \ Save$, $PIE$, and $MCP$.

$Jup/RS$ is calculated as the time that Jupiter and RedSpot would take to execute the number of instructions executed when not using PIE. It is the amount of time taken without PIE times the number of bytecodes executed with PIE and divided by the number of bytecodes executed without PIE. This calculation is an estimate of what the actual $Jup/RS$ time should be based on the assumption that the average time per instruction between executions is the same. $State \ Save$ is the amount of time taken by Jupiter and RedSpot when using PIE minus the estimate of time taken without using PIE. This difference is the amount of overhead incurred by Jupiter and RedSpot in saving information for PIE. $PIE$ is the amount of time for PIE to execute, and $MCP$ is the amount of time for MCP to perform its scheduling of 24 different configurations.

Figure A.1 contains the number of bytecodes that are executed for each benchmark for the five runs. The y-axis is logarithmic. The number of bytecodes executed is shown since it is used to distinguish between the execution of Jupiter and RedSpot and the saving of state.

Figure A.2 contains the simulation time for the runs that have window sizes of 128, 1024, and 8192. The y-axis for all three graphs is the percentage of time spent in each of the different parts of simulation relative to the total simulation time. The number at the top of each benchmark is the number of seconds of wallclock time that each execution took.
Figure A.1: Bytecodes executed.

For the 128 and 1024 window sizes, both MCP and PIE each take approximately 40% of the time, while MCP dominates the simulation time when the window size is 8192. The reason for this behaviour is that MCP’s execution time is proportional to the window size while the execution time of the other components is independent of window size.

If $n$ is the total number of traces executed and $w$ is the window size, then MCP is called $O(n/w)$ times and each invocation takes $O(w^2)$ time\(^1\). Therefore, the total time complexity is $O(nw)$. The work of the other components is based on the total number of traces processed, which is $O(n)$.

The importance of amortization is also visible in these graphs. With amortization a single execution of Jupiter with RedSpot and PIE results in twenty four different configurations being simulated. Without amortization the non MCP parts of simulation would take twenty four times more time and would cause simulation time to increase significantly. Instead, with amortization, simulation time is relatively balanced between the different parts and tends to be dominated by MCP for the larger window size.

\(^1\)According to published work [76] the complexity for each invocation of MCP is $O(w^2 \log(w))$ while an online draft of a more recent paper [75] states that the complexity is $O(w^2)$. The latter complexity is used here.
Appendix A. The Potential of Trace-Level Parallelism

Figure A.2: Simulation time.
A.2 Complete Performance Results

For completeness, the graphs that show the effects of communication type and cost for all combinations of 2, 4, or 8 processors and 128, 1024, or 8192 trace window sizes are presented. Figure A.3 contains the graphs for a window of size 128, Figure A.4 contains the graphs for a window of size 1024, and Figure A.5 contains the graphs for a window of size 8192. These figures show that the data is robust across a variety of configurations and therefore add support to our conclusions.
Figure A.3: Effect of the communication type for a window of size 128.
Figure A.4: Effect of the communication type for a window of size 1024.
Appendix A. The Potential of Trace-Level Parallelism

Figure A.5: Effect of the communication type for a window of size 8192.
Appendix B

Trace Collection Approaches

A trace collection system monitors program execution to identify when to start traces, records traces, identifies when to stop traces, and allows the recorded traces to be accessed. Each of these four actions need to be performed efficiently for overheads to be low. The most performance critical of these actions is the first one, monitoring program execution to identify of when to start traces. The reason is that the action occurs much more frequently than the other three actions. The occurrences of recording traces, deciding to stop a trace, and accessing recorded traces are proportional to the number of traces recorded. However, the occurrence of deciding whether to start recording or not is proportional to the number of instructions executed while the trace collection system is active. Therefore, reducing the amount of time spent making the decision will have the largest impact on reducing the overhead of trace collection.

One possible way to reduce the overhead is to use sampling, and only occasionally enable trace collection [78]. However, such an approach is less applicable in an online system because sampling collects less information for the execution of a given number of instructions. Instead, an online system would want to identify traces as quickly as possible for a portion of a program and then disable trace collection and perform optimization on that portion. Thus the overhead would be eliminated completely and the benefit of
optimization is realized as soon as possible. Only if the behaviour of that portion of the program would change significantly, would more trace collection be needed, in which case sampling may be useful.

An alternate approach, and the one we examine, is to change the criteria used to decide whether to start to record a trace or not. In particular, we examine the effect of not using trace exit events to start recording traces. The benefit is much simplified logic to decide whether to start recording a trace or not.

The following are brief descriptions of the implementations of trace collection systems with and without using trace exit events to decide whether to start recording traces.

B.1 With Trace Exit Events

Each control flow instruction and target are assigned a unique identifier. Counters, which start at zero and are incremented until they reach a threshold, are accessed through this unique identifier. Furthermore, data structures are maintained that keep track of whether execution is or is not on a trace. If execution is on a trace then the location within this trace that is being executed is also kept track of.

The baseline compiler inserts a method call into the trace collection system at each control flow instruction once its target is known. The method evaluates what is being executed and the state of trace execution. If a trace is not being recorded, then tests are performed to determine whether a trace is being executed or not, if a trace exit occurred, and if any event counters should be incremented. If a trace is being recorded then the instructions being executed are kept track of, and tests for when trace recording stops are performed.

In the fastest and most common case, when no recording occurs, the appropriate counter must be updated and four easy to predict and either three (when execution is on trace) or four (when execution is off trace) hard to predict branches must be executed.
Appendix B. Trace Collection Approaches

Since these operations must be performed very frequently, the overhead is large. Although the performance may be improved somewhat, there is an inherent limitation in that significant logic is required to keep track of trace execution.

B.2 Without Trace Exit Events

Each control flow instruction and target are assigned a unique identifier. Counters, which are initialized at the threshold and decremented until they reach zero, are accessed through this unique identifier. Furthermore, there is a recording variable that indicates whether a trace is being recorded or not.

If a counter is no longer being used, then it is set to a very large integer, thus ensuring that it will not execute. Although the counters could overflow, each unique identifier also has a flag that states if it is processed or not. If it is processed, then it is not processed again. Although there is extra overhead because this flag is only tested when recording is performed, no spurious recording takes place.

The baseline compiler inserts instructions at each control flow instruction once its target is known. The instructions check if the recording variable is set, decrement the appropriate counter, and check if recording should start because a counter reached zero, indicating that an event occurred. If recording should be performed, then a method is called that sets the counter to a very large number, checks the processed flag, adds the execution to the recorded trace, checks if recording should stop, and once recording stops, saves the trace.

The overhead is kept small by making the common case fast. Not using trace exits decreases the logic to only two easy to predict branches and one counter update. The five x86 assembly instructions that comprise the common case are a test to determine if the recording variable is set, a conditional jump that would go to the recording method if the recording variable is set, a move of the location of the counter referenced by the
identifier to a register, a decrement of the counter, and a conditional jump that skips the recording method if the counter has not passed zero. These instructions are followed by instructions for calling the method that records traces and then the rest of the program. The result is a very efficient trace collection system.

\section*{B.3 Trace Collection Evaluation}

Figure B.1 shows the execution time when only the baseline compiler is used. The y-axis is the speedup relative to the Jikes RVM using only the baseline compiler. The x-axis contains the benchmarks. The data points correspond to the Jikes RVM using the baseline compiler while still having counters keep track of what is frequently executed, using a trace collection system with trace exit events (and printing traces), using a trace collection system without trace exit events, and using a trace collection system without trace exit events that does not keep track of forward branches. These configurations are referred to as \textit{Baseline with counters}, \textit{TCS}, \textit{TCS2}, and \textit{TCS2 without forward branches}. The threshold is the same as in Chapter 8.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure_b_1.png}
\caption{Whole program information collection.}
\end{figure}
The results indicate that not using trace exit events can lead to significant performance improvement. The two TCS2 configurations have on average a speedup of 0.9, which compares well to the TCS configuration. Unfortunately, we have found that only using the baseline compiler leads to execution times that can vary considerably and unpredictably when small changes are made. For example, for bisort keeping track of forward branches leads to faster execution than not keeping track of forward branches. Therefore, performing more work leads to better performance. We leave investigating the reason for this behaviour for future work.

Figure B.2 shows the performance when runtime information is used to recompile frequently executed methods by the optimizing compiler. The y-axis is the speedup relative to the default configuration. The x-axis contains the benchmarks. The data points correspond to using trace exit events and optimizing at level O1; using trace exit events and optimizing at level O2; not using trace exit events and optimizing at level O1; not using trace exit events, not keeping track of forward branches, and optimizing at level O1; and not using trace exit events and optimizing at level O2. These configurations are referred to as TCS, TCS O2, TCS2, TCS2 without forward branches, and TCS2 O2. The trace collection systems optimize the top method of each trace.

The results indicate that using a more efficient trace collection system does improve the execution time, since all the TCS2 configurations are faster than the TCS configurations. However, the performance is still more than 10% lower than that of the default configuration on average, and for some benchmarks, the execution time is twice as slow as the default configuration. Therefore, good strategies need to be identified for a trace collection system to choose what to optimize. Although the current trace collection system that does not keep track of trace exit events is efficient enough, it needs to have a better strategy to decide what to optimize. We leave creating such a strategy for future work.
Figure B.2: Trace collection with recompilation.
Appendix C

Complete Timing Data for the Jikes RVM

This appendix contains a description of extracting and packaging a single trace as well as complete timing data for the results obtained using the Jikes RVM. The results are divided into two sections. The first contains an earlier study that looks at the effect on loop-based benchmarks of not grouping traces together [10]. The second contains all the wallclock times for running various configurations of the Jikes RVM.

C.1 Extracting and Packaging a Trace

Individual traces can be extracted and packaged. The steps are the same as for tasks, although the trace can only execute sequentially and instead of a fork and join, there is only a call to the trace. Figure C.1 contains an example of a trace being extracted. The left side of Figure C.1 has a method with an if statement and one trace that contains \( BB_1, BB_2, BB_4, BB_5, \) and \( BB_6 \). The right side has the methods that are the result of our transformation. The first step is the identification of the five blocks on the trace, and the edges \((BB_2, BB_3)\) and \((BB_6, \text{Exit})\) as trace exits. The second step is the creation of the new method with the basic blocks of the trace. The third step is the creation of
an entry block and two exit blocks in the new method. The fourth step is to replace $BB1$ in the original method with instructions that call the new method, check the return value to determine the exit that was taken and then jump either to $BB3$ or to $Exit$. $BB4$ is removed because it is not reachable in the original method. The fifth step is to add instructions that save and restore variables $a$, $b$, and $c$ in the original method. The variables $a$ and $b$ are saved and $c$ is updated. Conversely, the sixth step is to add instructions in the new method that restore the variables $a$ and $b$, and save variable $c$. After these six steps are performed, the trace is in its own method.

![Figure C.1: Extracting and packaging a trace.](image)

### C.2 Effects of not Grouping Traces

Figure C.2 contains the normalized execution time for the three loop-based programs on one processor when individual traces and groups of traces are extracted and packaged.
The individual traces cannot execute in parallel. The x-axis contains the normalized execution time. The execution time is normalized with respect to the execution time of the applications on the default configuration\(^1\). The x-axis contains three benchmarks, moldyn, montecarlo, and euler. The data points correspond to traces being extracted and packaged individually and in groups. These are referred to as traces and tasks, respectively. The figure shows that extracting and packaging individual traces introduces considerable overheads. For example, the execution time for moldyn is 33 times larger than for the adaptive configuration. Therefore, grouping traces into tasks is necessary.

![Figure C.2: Performance on one processor.](image)

C.3 Tables of Results

In this section, we describe our naming convention for configurations, show that using the default configuration to calculate speedup is appropriate, and present tables of all wallclock times.

\(^1\)The execution times are from early experiments and are not shown in the appendix. The data points are calculated based on single executions of the benchmarks.
Appendix C. Complete Timing Data for the Jikes RVM

The following is a brief description of configurations and suffixes that connects the configuration names in the table to what the Jikes RVM performs. The configurations can be divided into three parts: those that represent the Jikes RVM without trace collection or parallelization, the Jikes RVM with trace collection, and the Jikes RVM with parallelization. For all configurations, the final suffix represents the number of processors used\(^2\).

By default the Jikes RVM uses sampling to identify what to optimize and adaptively chooses which methods to optimize and which optimization level to use. This default configuration is referred to as default. The time suffix indicates that individual parts of the execution are timed. The overhead tends to be small, although it is usually non-zero. The baseline configuration uses only the baseline compiler and no recompilation can occur. This configuration disables the adaptive system. The baseline-counters configuration uses the baseline compiler and also maintains counters to keep track of what should be optimized, although the information is not used. The counters configuration uses counters instead of sampling. The optimizing configuration uses only the optimizing compiler, and not the baseline compiler. The adaptive system can still chose the level of optimization unless the once suffix is specified. The once suffix states that a method can only be compiled once. The optimization level of the compiler is 00, unless specified by the suffix 01 or 02, which specify different optimization levels (01 and 02 respectively)\(^3\).

Two different trace collection systems are considered, the one used in the body of the thesis, and a more efficient one described in Appendix B. The tcs configuration has the trace collection system used in the body of the thesis enabled. If only the baseline compiler is used then a baseline suffix is added, and otherwise, if the adaptive system can

---

\(^2\)The configurations without parallelization all use one processor. Although data is not shown here, we have found that there is no significant impact on performance of using more than one processor and that using one processor is at least as fast on average as using more processors.

\(^3\)Global code placement, which is enabled at optimization level 02 has been disabled when 02 is always used by the compiler because of errors that are reported in the loop invariant code motion optimization phase.
optimize methods, then an *adaptive* suffix is added. Furthermore, the trace collection system can either produce no output, produce traces, or produce tasks. The tasks are produced directly from the bytecode information of traces and therefore the mapping to the intermediate representation is not required. The three choices of outputs correspond to the prefixes *noout*, *traceout*, and *taskout* respectively. The *tcs2* configuration has the more efficient trace collection system described in Appendix B enabled. No output is produced by this trace collection system. If only the baseline compiler is used then a *baseline* suffix is added, and otherwise, if the adaptive system can optimize methods, then an *adaptive* suffix is added. Furthermore, if the *backward* suffix is added then forward branches do not have extra code that interfaces to the trace collection system added to them, and otherwise all branches have the extra code added. Although the wallclock times indicate that the relationship between where the instructions are added and performance is nontrivial, we leave the examination of the relationship for future work.

The parallelization of grouped traces is considered under many different configurations. The starting configuration is referred to as *trace*, and represents the parallelization system when the system is not notified of when the worker threads will become active, when synchronization is used to determine whether worker threads are active, when the main thread yields if it is waiting on worker threads to complete work, and when the maximum nesting level is three. The *soonactive* suffix indicates that the system is notified when the worker threads will become active. The *noyield* suffix indicates that the main thread does not yield when it is waiting for worker threads to complete their work. Furthermore, the *min* suffix indicates that loop-based tasks only execute in parallel if the number of iterations is greater than the number after *min*. Having a minimum number of iterations prevents parallelization overheads to be incurred for loops that perform almost no work. Also, the *thrd* suffix indicates that the worker threads wait a small amount of time before yielding when they have no work. The time is proportional to the number
after \textit{thrd}, which indicates the number of iterations of a simple loop. The last two suffixes are used to increase the responsiveness of communication between the main and worker threads at the expense of slowing down the JVM. These four suffixes directly affect loop-based tasks. The suffix \textit{twolevel} indicates a maximum nesting level of two. The suffix \textit{fourlevel} indicates a maximum nesting level of four. Both these suffixes directly affect recursive tasks. The \textit{nosync} suffix indicates that a synchronization block is not used to check whether a worker thread is active or not. Instead, only the fact that the variable being checked is volatile is used to ensure proper execution. This suffix affects all tasks. The \textit{trace-noyield-soonactive-once-nosync-O2-min50-thrd3200-4} configuration failed to execute for Perimeter because of a bad interaction between the garbage collector and the prototype.

Figure C.3 (page 241) shows the various configurations of the Jikes RVM with the trace collection systems and the automatic parallelization system disabled. The y-axis is the speedup relative to the default configuration and the x-axis contains the benchmarks along with the geometric mean. The data points correspond to the different configurations with all methods chosen by the adaptive system optimized at \textit{O2}, at \textit{O1}, at \textit{O0}, when the initial compiler is set to be the optimizing compiler instead of the baseline compiler, and when counters are used instead of sampling. These configurations are referred to as \textit{O2}, \textit{O1}, \textit{O0}, \textit{optimizing}, and \textit{counters}, respectively. The results show that the default configuration is the fastest overall. Therefore, the results validate our choice of the default configuration as the one that should be used to calculate speedup.

Tables C.1 through C.7 contain the execution times for the various configurations. The data can be used to get precise information regarding the execution times of the different benchmarks when different configurations are used.
Appendix C. Complete Timing Data for the Jikes RVM

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline-1</td>
<td>47.39</td>
<td>47.39</td>
<td>47.41</td>
<td>47.42</td>
<td>47.42</td>
</tr>
<tr>
<td>baseline-counters-1</td>
<td>176.47</td>
<td>176.35</td>
<td>176.57</td>
<td>176.21</td>
<td>176.34</td>
</tr>
<tr>
<td>counters-1</td>
<td>18.97</td>
<td>18.95</td>
<td>18.97</td>
<td>18.93</td>
<td>18.92</td>
</tr>
<tr>
<td>default-1</td>
<td>20.00</td>
<td>20.02</td>
<td>20.02</td>
<td>19.96</td>
<td>20.17</td>
</tr>
<tr>
<td>default-time-1</td>
<td>20.07</td>
<td>20.09</td>
<td>20.07</td>
<td>20.06</td>
<td>20.02</td>
</tr>
<tr>
<td>trace-noyield-nosync-min50-thrd3200-4</td>
<td>12.42</td>
<td>12.49</td>
<td>12.41</td>
<td>12.39</td>
<td>12.51</td>
</tr>
<tr>
<td>trace-noyield-soonactive-min50-thrd3200-4</td>
<td>12.27</td>
<td>12.29</td>
<td>12.32</td>
<td>12.42</td>
<td>12.32</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-4</td>
<td>12.29</td>
<td>12.28</td>
<td>12.29</td>
<td>12.30</td>
<td>12.31</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-1</td>
<td>20.77</td>
<td>20.74</td>
<td>20.80</td>
<td>20.81</td>
<td>20.73</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-3</td>
<td>13.18</td>
<td>13.29</td>
<td>13.16</td>
<td>13.21</td>
<td>13.27</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-4</td>
<td>12.19</td>
<td>12.22</td>
<td>12.25</td>
<td>12.20</td>
<td>12.28</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min100-thrd3200-4</td>
<td>12.31</td>
<td>12.30</td>
<td>12.31</td>
<td>12.29</td>
<td>12.42</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-4</td>
<td>12.44</td>
<td>12.43</td>
<td>12.30</td>
<td>12.45</td>
<td>12.40</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-1</td>
<td>20.51</td>
<td>20.56</td>
<td>20.56</td>
<td>20.57</td>
<td>20.57</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-3</td>
<td>14.50</td>
<td>14.64</td>
<td>14.60</td>
<td>14.43</td>
<td>14.63</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-4</td>
<td>12.31</td>
<td>12.42</td>
<td>12.32</td>
<td>12.33</td>
<td>12.34</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-thrd3200-4</td>
<td>12.28</td>
<td>12.28</td>
<td>12.29</td>
<td>12.34</td>
<td>12.35</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-1</td>
<td>20.65</td>
<td>20.65</td>
<td>20.63</td>
<td>20.62</td>
<td>20.66</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-4</td>
<td>12.39</td>
<td>12.42</td>
<td>12.40</td>
<td>12.41</td>
<td>12.44</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-1</td>
<td>20.77</td>
<td>20.74</td>
<td>20.76</td>
<td>20.80</td>
<td>21.04</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-2</td>
<td>15.15</td>
<td>15.10</td>
<td>15.11</td>
<td>15.15</td>
<td>15.12</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-3</td>
<td>15.36</td>
<td>15.35</td>
<td>15.34</td>
<td>15.33</td>
<td>15.33</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-4</td>
<td>15.43</td>
<td>15.39</td>
<td>15.46</td>
<td>15.41</td>
<td>15.42</td>
</tr>
<tr>
<td>trace-noyield-soonactive-once-nosync-O1-min50-thrd3200-4</td>
<td>13.34</td>
<td>13.33</td>
<td>13.49</td>
<td>13.32</td>
<td>13.34</td>
</tr>
<tr>
<td>trace-noyield-soonactive-once-nosync-min50-thrd3200-4</td>
<td>13.33</td>
<td>13.31</td>
<td>13.36</td>
<td>13.33</td>
<td>13.34</td>
</tr>
<tr>
<td>trace-soonactive-nosync-min50-thrd3200-4</td>
<td>12.64</td>
<td>12.29</td>
<td>12.32</td>
<td>12.28</td>
<td>12.33</td>
</tr>
<tr>
<td>optimizing-1</td>
<td>20.81</td>
<td>21.03</td>
<td>20.85</td>
<td>20.83</td>
<td>20.81</td>
</tr>
<tr>
<td>optimizing-once-1</td>
<td>22.00</td>
<td>22.03</td>
<td>22.00</td>
<td>22.00</td>
<td>22.03</td>
</tr>
<tr>
<td>optimizing-once-O1-1</td>
<td>22.02</td>
<td>22.06</td>
<td>22.01</td>
<td>22.03</td>
<td>22.01</td>
</tr>
<tr>
<td>optimizing-once-O2-1</td>
<td>20.85</td>
<td>20.84</td>
<td>20.83</td>
<td>20.86</td>
<td>20.84</td>
</tr>
<tr>
<td>tcs-adaptive-O2-noout-1</td>
<td>23.59</td>
<td>25.07</td>
<td>23.27</td>
<td>24.98</td>
<td>23.60</td>
</tr>
<tr>
<td>tcs-adaptive-noout-1</td>
<td>19.01</td>
<td>19.03</td>
<td>19.06</td>
<td>20.93</td>
<td>19.03</td>
</tr>
<tr>
<td>tcs-baseline-noout-1</td>
<td>196.01</td>
<td>194.96</td>
<td>195.03</td>
<td>194.90</td>
<td>197.51</td>
</tr>
<tr>
<td>tcs-baseline-tasksout-1</td>
<td>148.80</td>
<td>149.01</td>
<td>148.76</td>
<td>148.81</td>
<td>148.85</td>
</tr>
<tr>
<td>tcs-baseline-tracesout-1</td>
<td>148.69</td>
<td>148.70</td>
<td>148.97</td>
<td>148.68</td>
<td>148.88</td>
</tr>
<tr>
<td>tcs2-adaptive-1</td>
<td>19.03</td>
<td>19.06</td>
<td>19.05</td>
<td>19.02</td>
<td>19.06</td>
</tr>
<tr>
<td>tcs2-adaptive-backward-1</td>
<td>19.01</td>
<td>18.97</td>
<td>19.03</td>
<td>19.03</td>
<td>19.01</td>
</tr>
<tr>
<td>tcs2-baseline-1</td>
<td>64.58</td>
<td>64.51</td>
<td>64.52</td>
<td>64.50</td>
<td>64.46</td>
</tr>
<tr>
<td>tcs2-baseline-backward-1</td>
<td>119.95</td>
<td>119.71</td>
<td>119.81</td>
<td>119.81</td>
<td>119.78</td>
</tr>
</tbody>
</table>

Table C.1: Wallclock times for BiSort.
## Appendix C. Complete Timing Data for the Jikes RVM

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline-1</td>
<td>55.47</td>
<td>55.45</td>
<td>55.38</td>
<td>55.41</td>
<td>55.40</td>
</tr>
<tr>
<td>baseline-counters-1</td>
<td>53.28</td>
<td>53.30</td>
<td>53.40</td>
<td>53.31</td>
<td>53.31</td>
</tr>
<tr>
<td>counters-1</td>
<td>5.10</td>
<td>5.12</td>
<td>5.11</td>
<td>5.11</td>
<td>5.10</td>
</tr>
<tr>
<td>default-1</td>
<td>5.51</td>
<td>5.56</td>
<td>5.63</td>
<td>5.67</td>
<td>5.39</td>
</tr>
<tr>
<td>default-time-1</td>
<td>5.83</td>
<td>5.39</td>
<td>5.55</td>
<td>5.62</td>
<td>5.76</td>
</tr>
<tr>
<td>trace-noyield-nosync-min50-thrd3200-4</td>
<td>2.93</td>
<td>3.21</td>
<td>3.28</td>
<td>3.24</td>
<td>2.98</td>
</tr>
<tr>
<td>trace-noyield-soonactive-min50-thrd3200-4</td>
<td>2.20</td>
<td>2.19</td>
<td>2.16</td>
<td>2.26</td>
<td>2.17</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-4</td>
<td>2.23</td>
<td>2.21</td>
<td>2.22</td>
<td>2.26</td>
<td>2.32</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-1</td>
<td>10.28</td>
<td>10.26</td>
<td>10.27</td>
<td>10.26</td>
<td>10.40</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-2</td>
<td>25.64</td>
<td>25.98</td>
<td>25.80</td>
<td>6.15</td>
<td>25.61</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-3</td>
<td>5.36</td>
<td>5.24</td>
<td>5.24</td>
<td>5.24</td>
<td>5.34</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-4</td>
<td>6.45</td>
<td>6.87</td>
<td>7.45</td>
<td>6.04</td>
<td>6.12</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min100-thrd3200-4</td>
<td>2.47</td>
<td>2.24</td>
<td>2.20</td>
<td>2.22</td>
<td>2.23</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-4</td>
<td>2.23</td>
<td>2.22</td>
<td>2.21</td>
<td>2.26</td>
<td>2.31</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-1</td>
<td>5.18</td>
<td>5.17</td>
<td>5.17</td>
<td>5.17</td>
<td>5.16</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-3</td>
<td>2.53</td>
<td>2.51</td>
<td>2.56</td>
<td>2.57</td>
<td>2.54</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-4</td>
<td>2.27</td>
<td>2.23</td>
<td>2.19</td>
<td>2.29</td>
<td>2.21</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-thrd3200-4</td>
<td>2.35</td>
<td>2.24</td>
<td>2.33</td>
<td>2.48</td>
<td>2.21</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-1</td>
<td>5.22</td>
<td>5.20</td>
<td>5.22</td>
<td>5.23</td>
<td>5.22</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-2</td>
<td>3.18</td>
<td>3.16</td>
<td>3.16</td>
<td>3.15</td>
<td>3.19</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-3</td>
<td>2.58</td>
<td>2.63</td>
<td>2.55</td>
<td>2.58</td>
<td>2.58</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-4</td>
<td>2.30</td>
<td>2.33</td>
<td>2.24</td>
<td>2.40</td>
<td>2.27</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-1</td>
<td>5.17</td>
<td>5.09</td>
<td>5.07</td>
<td>5.16</td>
<td>5.07</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-2</td>
<td>3.11</td>
<td>3.06</td>
<td>3.08</td>
<td>3.05</td>
<td>3.13</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-3</td>
<td>3.10</td>
<td>3.11</td>
<td>3.10</td>
<td>3.12</td>
<td>3.08</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-4</td>
<td>2.10</td>
<td>2.09</td>
<td>2.08</td>
<td>2.12</td>
<td>2.08</td>
</tr>
<tr>
<td>trace-noyield-soonactive-once-nosync-O1-min50-thrd3200-4</td>
<td>2.27</td>
<td>2.38</td>
<td>2.32</td>
<td>2.28</td>
<td>2.24</td>
</tr>
<tr>
<td>trace-noyield-soonactive-once-nosync-O2-min50-thrd3200-4</td>
<td>2.96</td>
<td>2.94</td>
<td>3.10</td>
<td>2.94</td>
<td>3.16</td>
</tr>
<tr>
<td>trace-noyield-soonactive-once-nosync-min50-thrd3200-4</td>
<td>2.21</td>
<td>2.29</td>
<td>2.30</td>
<td>2.22</td>
<td>2.23</td>
</tr>
<tr>
<td>trace-soonactive-nosync-min50-thrd3200-4</td>
<td>2.20</td>
<td>2.32</td>
<td>2.25</td>
<td>2.28</td>
<td>2.39</td>
</tr>
<tr>
<td>optimizing-1</td>
<td>5.30</td>
<td>5.31</td>
<td>5.30</td>
<td>5.31</td>
<td>5.31</td>
</tr>
<tr>
<td>optimizing-once-1</td>
<td>5.34</td>
<td>5.33</td>
<td>5.33</td>
<td>5.33</td>
<td>5.34</td>
</tr>
<tr>
<td>optimizing-once-O1-1</td>
<td>5.30</td>
<td>5.32</td>
<td>5.31</td>
<td>5.31</td>
<td>5.30</td>
</tr>
<tr>
<td>optimizing-once-O2-1</td>
<td>5.58</td>
<td>5.59</td>
<td>5.59</td>
<td>5.58</td>
<td>5.58</td>
</tr>
<tr>
<td>tcs-adaptive-O2-noout-1</td>
<td>20.77</td>
<td>23.34</td>
<td>23.76</td>
<td>23.18</td>
<td>21.50</td>
</tr>
<tr>
<td>tcs-adaptive-noout-1</td>
<td>23.69</td>
<td>19.26</td>
<td>19.49</td>
<td>19.00</td>
<td>21.66</td>
</tr>
<tr>
<td>tcs-baseline-noout-1</td>
<td>102.88</td>
<td>102.97</td>
<td>102.96</td>
<td>102.92</td>
<td>102.89</td>
</tr>
<tr>
<td>tcs-baseline-tasksout-1</td>
<td>89.01</td>
<td>89.04</td>
<td>89.19</td>
<td>89.07</td>
<td>89.04</td>
</tr>
<tr>
<td>tcs-baseline-tracesout-1</td>
<td>88.95</td>
<td>89.02</td>
<td>88.92</td>
<td>88.99</td>
<td>88.91</td>
</tr>
<tr>
<td>tcs2-adaptive-1</td>
<td>11.58</td>
<td>10.22</td>
<td>10.43</td>
<td>11.60</td>
<td>10.26</td>
</tr>
<tr>
<td>tcs2-adaptive-O2-1</td>
<td>10.90</td>
<td>11.58</td>
<td>11.60</td>
<td>10.44</td>
<td>11.63</td>
</tr>
<tr>
<td>tcs2-adaptive-backardal-1</td>
<td>14.84</td>
<td>14.88</td>
<td>12.54</td>
<td>14.71</td>
<td>14.71</td>
</tr>
<tr>
<td>tcs2-baseline-1</td>
<td>46.46</td>
<td>46.55</td>
<td>46.48</td>
<td>46.51</td>
<td>46.43</td>
</tr>
<tr>
<td>tcs2-baseline-backward-1</td>
<td>52.94</td>
<td>52.93</td>
<td>52.92</td>
<td>52.91</td>
<td>52.91</td>
</tr>
</tbody>
</table>

Table C.2: Wallclock times for Perimeter.
<table>
<thead>
<tr>
<th>Configuration</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline-1</td>
<td>10.29</td>
<td>10.28</td>
<td>10.28</td>
<td>10.28</td>
<td>10.28</td>
</tr>
<tr>
<td>baseline-counters-1</td>
<td>10.42</td>
<td>10.42</td>
<td>10.43</td>
<td>10.41</td>
<td>10.42</td>
</tr>
<tr>
<td>counters-1</td>
<td>1.62</td>
<td>1.62</td>
<td>1.62</td>
<td>1.62</td>
<td>1.62</td>
</tr>
<tr>
<td>default-1</td>
<td>1.83</td>
<td>1.83</td>
<td>1.83</td>
<td>1.82</td>
<td>1.82</td>
</tr>
<tr>
<td>default-time-1</td>
<td>1.86</td>
<td>1.87</td>
<td>1.87</td>
<td>1.86</td>
<td>1.87</td>
</tr>
<tr>
<td>trace-noyield-nosync-min50-thrd3200-4</td>
<td>1.48</td>
<td>1.46</td>
<td>1.47</td>
<td>1.45</td>
<td>1.46</td>
</tr>
<tr>
<td>trace-noyield-soonactive-min50-thrd3200-4</td>
<td>1.44</td>
<td>1.45</td>
<td>1.45</td>
<td>1.45</td>
<td>1.46</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-4</td>
<td>1.45</td>
<td>1.46</td>
<td>1.46</td>
<td>1.46</td>
<td>1.44</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-1</td>
<td>1.93</td>
<td>1.93</td>
<td>1.93</td>
<td>1.93</td>
<td>1.94</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-2</td>
<td>1.63</td>
<td>1.62</td>
<td>1.62</td>
<td>1.64</td>
<td>1.67</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-3</td>
<td>1.55</td>
<td>1.54</td>
<td>1.55</td>
<td>1.55</td>
<td>1.57</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-4</td>
<td>1.59</td>
<td>1.56</td>
<td>1.55</td>
<td>1.60</td>
<td>1.56</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min100-thrd3200-4</td>
<td>1.44</td>
<td>1.45</td>
<td>1.45</td>
<td>1.48</td>
<td>1.44</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-4</td>
<td>1.45</td>
<td>1.44</td>
<td>1.44</td>
<td>1.45</td>
<td>1.44</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-1</td>
<td>1.92</td>
<td>1.92</td>
<td>1.94</td>
<td>1.92</td>
<td>1.92</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-2</td>
<td>1.56</td>
<td>1.56</td>
<td>1.56</td>
<td>1.56</td>
<td>1.56</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-3</td>
<td>1.60</td>
<td>1.60</td>
<td>1.58</td>
<td>1.60</td>
<td>1.61</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-4</td>
<td>1.46</td>
<td>1.44</td>
<td>1.46</td>
<td>1.46</td>
<td>1.43</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-1</td>
<td>1.98</td>
<td>1.99</td>
<td>1.97</td>
<td>1.98</td>
<td>1.97</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-2</td>
<td>1.61</td>
<td>1.61</td>
<td>1.61</td>
<td>1.61</td>
<td>1.61</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-3</td>
<td>1.65</td>
<td>1.65</td>
<td>1.65</td>
<td>1.64</td>
<td>1.64</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-4</td>
<td>1.49</td>
<td>1.50</td>
<td>1.49</td>
<td>1.49</td>
<td>1.50</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-1</td>
<td>1.94</td>
<td>1.94</td>
<td>1.94</td>
<td>1.94</td>
<td>1.94</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-2</td>
<td>1.54</td>
<td>1.56</td>
<td>1.56</td>
<td>1.56</td>
<td>1.55</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-3</td>
<td>1.60</td>
<td>1.59</td>
<td>1.59</td>
<td>1.61</td>
<td>1.59</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-4</td>
<td>1.66</td>
<td>1.68</td>
<td>1.68</td>
<td>1.65</td>
<td>1.66</td>
</tr>
<tr>
<td>trace-noyield-soonactive-once-nosync-O1-min50-thrd3200-4</td>
<td>1.46</td>
<td>1.46</td>
<td>1.45</td>
<td>1.49</td>
<td>1.45</td>
</tr>
<tr>
<td>trace-noyield-soonactive-once-nosync-O2-min50-thrd3200-4</td>
<td>1.82</td>
<td>1.80</td>
<td>1.79</td>
<td>1.81</td>
<td>1.79</td>
</tr>
<tr>
<td>trace-noyield-soonactive-once-nosync-min50-thrd3200-4</td>
<td>1.47</td>
<td>1.45</td>
<td>1.45</td>
<td>1.49</td>
<td>1.45</td>
</tr>
<tr>
<td>trace-soonactive-nosync-min50-thrd3200-4</td>
<td>1.45</td>
<td>1.45</td>
<td>1.44</td>
<td>1.46</td>
<td>1.46</td>
</tr>
<tr>
<td>optimizing-1</td>
<td>1.74</td>
<td>1.74</td>
<td>1.74</td>
<td>1.73</td>
<td>1.74</td>
</tr>
<tr>
<td>optimizing-once-1</td>
<td>1.75</td>
<td>1.74</td>
<td>1.74</td>
<td>1.74</td>
<td>1.74</td>
</tr>
<tr>
<td>optimizing-once-O1-1</td>
<td>1.75</td>
<td>1.75</td>
<td>1.76</td>
<td>1.74</td>
<td>1.75</td>
</tr>
<tr>
<td>optimizing-once-O2-1</td>
<td>1.94</td>
<td>1.94</td>
<td>1.93</td>
<td>1.93</td>
<td>1.93</td>
</tr>
<tr>
<td>tcs-adaptive-O2-noout-1</td>
<td>1.70</td>
<td>1.70</td>
<td>1.70</td>
<td>1.73</td>
<td>1.70</td>
</tr>
<tr>
<td>tcs-adaptive-noout-1</td>
<td>1.73</td>
<td>1.71</td>
<td>1.70</td>
<td>1.73</td>
<td>1.68</td>
</tr>
<tr>
<td>tcs-baseline-noout-1</td>
<td>17.10</td>
<td>17.11</td>
<td>17.10</td>
<td>17.11</td>
<td>17.10</td>
</tr>
<tr>
<td>tcs-baseline-tasksout-1</td>
<td>15.29</td>
<td>15.27</td>
<td>15.29</td>
<td>15.28</td>
<td>15.27</td>
</tr>
<tr>
<td>tcs-baseline-tracesout-1</td>
<td>15.28</td>
<td>15.24</td>
<td>15.27</td>
<td>15.24</td>
<td>15.27</td>
</tr>
<tr>
<td>tcs2-adaptive-1</td>
<td>1.64</td>
<td>1.65</td>
<td>1.63</td>
<td>1.63</td>
<td>1.65</td>
</tr>
<tr>
<td>tcs2-adaptive-O2-1</td>
<td>1.62</td>
<td>1.60</td>
<td>1.61</td>
<td>1.62</td>
<td>1.60</td>
</tr>
<tr>
<td>tcs2-adaptive-backward-1</td>
<td>1.62</td>
<td>1.63</td>
<td>1.62</td>
<td>1.63</td>
<td>1.62</td>
</tr>
<tr>
<td>tcs2-baseline-1</td>
<td>11.36</td>
<td>11.36</td>
<td>11.37</td>
<td>11.36</td>
<td>11.37</td>
</tr>
<tr>
<td>tcs2-baseline-backward-1</td>
<td>7.64</td>
<td>7.66</td>
<td>7.64</td>
<td>7.65</td>
<td>7.64</td>
</tr>
</tbody>
</table>

Table C.3: Wallclock times for TreeAdd.
### Appendix C. Complete Timing Data for the Jikes RVM

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline-1</td>
<td>64.74</td>
<td>64.74</td>
<td>64.74</td>
<td>64.74</td>
<td>64.74</td>
</tr>
<tr>
<td>baseline-counters-1</td>
<td>66.21</td>
<td>66.21</td>
<td>66.20</td>
<td>66.21</td>
<td>66.24</td>
</tr>
<tr>
<td>counters-1</td>
<td>72.25</td>
<td>72.13</td>
<td>72.24</td>
<td>72.18</td>
<td>73.01</td>
</tr>
<tr>
<td>default-1</td>
<td>28.36</td>
<td>26.09</td>
<td>26.29</td>
<td>28.31</td>
<td>26.27</td>
</tr>
<tr>
<td>trace-noyield-nosync-min50-thrd3200-4</td>
<td>12.77</td>
<td>12.28</td>
<td>12.26</td>
<td>12.06</td>
<td>12.08</td>
</tr>
<tr>
<td>trace-noyield-soonactive-min50-thrd3200-4</td>
<td>11.92</td>
<td>11.98</td>
<td>13.01</td>
<td>19.41</td>
<td>11.96</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-4</td>
<td>11.99</td>
<td>19.35</td>
<td>12.37</td>
<td>17.54</td>
<td>12.32</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-1</td>
<td>31.75</td>
<td>31.69</td>
<td>31.87</td>
<td>31.93</td>
<td>31.65</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-3</td>
<td>15.75</td>
<td>15.82</td>
<td>15.52</td>
<td>15.28</td>
<td>15.76</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-4</td>
<td>12.39</td>
<td>16.83</td>
<td>13.05</td>
<td>12.42</td>
<td>12.88</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min100-thrd3200-4</td>
<td>19.74</td>
<td>12.23</td>
<td>16.58</td>
<td>11.99</td>
<td>12.16</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-4</td>
<td>12.26</td>
<td>11.98</td>
<td>19.53</td>
<td>12.05</td>
<td>12.04</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-1</td>
<td>31.25</td>
<td>31.15</td>
<td>31.21</td>
<td>31.24</td>
<td>31.14</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-2</td>
<td>18.11</td>
<td>18.43</td>
<td>18.35</td>
<td>18.50</td>
<td>18.20</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-3</td>
<td>17.55</td>
<td>17.28</td>
<td>17.40</td>
<td>17.55</td>
<td>16.99</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-thrd3200-4</td>
<td>12.12</td>
<td>12.16</td>
<td>12.09</td>
<td>12.10</td>
<td>12.09</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-1</td>
<td>31.13</td>
<td>30.81</td>
<td>30.74</td>
<td>30.87</td>
<td>30.99</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-2</td>
<td>18.59</td>
<td>17.88</td>
<td>18.10</td>
<td>18.40</td>
<td>17.86</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-3</td>
<td>17.24</td>
<td>17.67</td>
<td>17.63</td>
<td>17.11</td>
<td>17.91</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-4</td>
<td>12.56</td>
<td>12.28</td>
<td>12.12</td>
<td>17.35</td>
<td>12.49</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-1</td>
<td>32.92</td>
<td>32.44</td>
<td>32.95</td>
<td>32.99</td>
<td>32.82</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-3</td>
<td>22.47</td>
<td>18.45</td>
<td>21.09</td>
<td>17.72</td>
<td>21.50</td>
</tr>
<tr>
<td>trace-noyield-soonactive- once-nosync-O1-min50-thrd3200-4</td>
<td>22.58</td>
<td>22.56</td>
<td>26.46</td>
<td>22.78</td>
<td>26.50</td>
</tr>
<tr>
<td>trace-noyield-soonactive- once-nosync-min50-thrd3200-4</td>
<td>22.66</td>
<td>22.57</td>
<td>22.71</td>
<td>28.32</td>
<td>23.68</td>
</tr>
<tr>
<td>trace-soonactive-nosync-min50-thrd3200-4</td>
<td>12.24</td>
<td>12.17</td>
<td>11.98</td>
<td>12.22</td>
<td>11.94</td>
</tr>
<tr>
<td>optimizing-1</td>
<td>32.44</td>
<td>32.44</td>
<td>32.33</td>
<td>32.16</td>
<td>32.53</td>
</tr>
<tr>
<td>optimizing-one-1</td>
<td>73.28</td>
<td>73.19</td>
<td>73.09</td>
<td>73.19</td>
<td>73.06</td>
</tr>
<tr>
<td>optimizing-one-O1-1</td>
<td>73.00</td>
<td>73.04</td>
<td>73.01</td>
<td>72.94</td>
<td>72.95</td>
</tr>
<tr>
<td>optimizing-one-O2-1</td>
<td>23.12</td>
<td>23.13</td>
<td>23.11</td>
<td>23.12</td>
<td>23.09</td>
</tr>
<tr>
<td>tcs-adaptive-noout-1</td>
<td>74.05</td>
<td>73.99</td>
<td>74.26</td>
<td>74.20</td>
<td>74.78</td>
</tr>
<tr>
<td>tcs-baseline-noout-1</td>
<td>92.61</td>
<td>92.57</td>
<td>92.62</td>
<td>92.62</td>
<td>92.59</td>
</tr>
<tr>
<td>tcs-baseline-tasksout-1</td>
<td>93.01</td>
<td>93.03</td>
<td>93.06</td>
<td>92.98</td>
<td>93.04</td>
</tr>
<tr>
<td>tcs-baseline-tracesout-1</td>
<td>92.92</td>
<td>92.92</td>
<td>92.94</td>
<td>92.93</td>
<td>93.02</td>
</tr>
<tr>
<td>tcs2-adaptive-1</td>
<td>73.70</td>
<td>73.51</td>
<td>73.03</td>
<td>73.99</td>
<td>73.44</td>
</tr>
<tr>
<td>tcs2-adaptive-O2-1</td>
<td>24.31</td>
<td>24.00</td>
<td>23.87</td>
<td>24.30</td>
<td>23.97</td>
</tr>
<tr>
<td>tcs2-adaptive-backward-1</td>
<td>74.17</td>
<td>72.76</td>
<td>74.06</td>
<td>74.25</td>
<td>74.09</td>
</tr>
<tr>
<td>tcs2-baseline-1</td>
<td>66.75</td>
<td>66.73</td>
<td>66.72</td>
<td>66.71</td>
<td>66.74</td>
</tr>
<tr>
<td>tcs2-baseline-backward-1</td>
<td>66.79</td>
<td>66.81</td>
<td>66.80</td>
<td>66.79</td>
<td>66.78</td>
</tr>
</tbody>
</table>

Table C.4: Wallclock times for TSP.
## Appendix C. Complete Timing Data for the Jikes RVM

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline-1</td>
<td>100.21</td>
<td>100.35</td>
<td>100.29</td>
<td>100.19</td>
<td>100.20</td>
</tr>
<tr>
<td>baseline-counters-1</td>
<td>101.93</td>
<td>101.92</td>
<td>101.90</td>
<td>101.91</td>
<td>101.91</td>
</tr>
<tr>
<td>counters-1</td>
<td>7.70</td>
<td>7.77</td>
<td>7.91</td>
<td>7.71</td>
<td>7.63</td>
</tr>
<tr>
<td>default-1</td>
<td>7.35</td>
<td>7.51</td>
<td>7.25</td>
<td>7.42</td>
<td>7.42</td>
</tr>
<tr>
<td>default-time-1</td>
<td>7.68</td>
<td>7.35</td>
<td>7.44</td>
<td>7.58</td>
<td>7.42</td>
</tr>
<tr>
<td>trace-nyield-nosync-min50-thrd3200-4</td>
<td>9.61</td>
<td>9.52</td>
<td>9.53</td>
<td>6.21</td>
<td>6.29</td>
</tr>
<tr>
<td>trace-nyield-soonactive-min50-thrd3200-4</td>
<td>10.05</td>
<td>9.98</td>
<td>10.09</td>
<td>18.21</td>
<td>10.13</td>
</tr>
<tr>
<td>trace-nyield-soonactive-nosync-min50-thrd3200-1</td>
<td>13.47</td>
<td>7.28</td>
<td>7.27</td>
<td>7.98</td>
<td>7.17</td>
</tr>
<tr>
<td>trace-nyield-soonactive-nosync-fourlevels-min50-thrd3200-2</td>
<td>6.46</td>
<td>10.99</td>
<td>6.34</td>
<td>6.29</td>
<td>6.34</td>
</tr>
<tr>
<td>trace-nyield-soonactive-nosync-fourlevels-min50-thrd3200-3</td>
<td>6.21</td>
<td>9.64</td>
<td>6.29</td>
<td>6.19</td>
<td>6.15</td>
</tr>
<tr>
<td>trace-nyield-soonactive-nosync-fivelevels-min50-thrd3200-4</td>
<td>37.60</td>
<td>40.37</td>
<td>42.97</td>
<td>38.04</td>
<td>36.82</td>
</tr>
<tr>
<td>trace-nyield-soonactive-nosync-min100-thrd3200-4</td>
<td>6.59</td>
<td>10.07</td>
<td>10.00</td>
<td>9.97</td>
<td>9.96</td>
</tr>
<tr>
<td>trace-nyield-soonactive-nosync-min50-4</td>
<td>38.32</td>
<td>38.25</td>
<td>44.00</td>
<td>44.28</td>
<td>38.88</td>
</tr>
<tr>
<td>trace-nyield-soonactive-nosync-min50-thrd3200-1</td>
<td>10.04</td>
<td>24.64</td>
<td>18.17</td>
<td>9.96</td>
<td>17.96</td>
</tr>
<tr>
<td>trace-nyield-soonactive-nosync-min50-thrd3200-2</td>
<td>7.27</td>
<td>13.60</td>
<td>7.26</td>
<td>8.13</td>
<td>14.31</td>
</tr>
<tr>
<td>trace-nyield-soonactive-nosync-fivelevels-min50-thrd3200-3</td>
<td>6.37</td>
<td>6.31</td>
<td>6.36</td>
<td>6.36</td>
<td>6.37</td>
</tr>
<tr>
<td>trace-nyield-soonactive-nosync-fivelevels-min50-thrd3200-4</td>
<td>6.25</td>
<td>6.45</td>
<td>6.18</td>
<td>6.28</td>
<td>6.24</td>
</tr>
<tr>
<td>trace-nyield-soonactive-nosync-time-fivelevels-min50-thrd3200-1</td>
<td>10.33</td>
<td>13.10</td>
<td>10.15</td>
<td>13.04</td>
<td>11.44</td>
</tr>
<tr>
<td>trace-nyield-soonactive-nosync-time-fivelevels-min50-thrd3200-2</td>
<td>7.73</td>
<td>7.78</td>
<td>7.58</td>
<td>8.67</td>
<td>8.68</td>
</tr>
<tr>
<td>trace-nyield-soonactive-nosync-time-fivelevels-min50-thrd3200-3</td>
<td>7.38</td>
<td>6.71</td>
<td>6.82</td>
<td>7.28</td>
<td>6.74</td>
</tr>
<tr>
<td>trace-nyield-soonactive-nosync-time-fivelevels-min50-thrd3200-4</td>
<td>7.09</td>
<td>6.70</td>
<td>6.99</td>
<td>6.93</td>
<td>7.00</td>
</tr>
<tr>
<td>trace-nyield-soonactive-nosync-twolevels-min50-thrd3200-1</td>
<td>18.33</td>
<td>10.01</td>
<td>18.27</td>
<td>9.88</td>
<td>10.15</td>
</tr>
<tr>
<td>trace-nyield-soonactive-nosync-twolevels-min50-thrd3200-3</td>
<td>6.32</td>
<td>10.89</td>
<td>10.92</td>
<td>6.34</td>
<td>6.31</td>
</tr>
<tr>
<td>trace-nyield-soonactive-nosync-twolevels-min50-thrd3200-4</td>
<td>9.64</td>
<td>9.70</td>
<td>6.36</td>
<td>6.48</td>
<td>6.18</td>
</tr>
<tr>
<td>trace-nyield-soonactive-once-nosync-O1-min50-thrd3200-4</td>
<td>9.69</td>
<td>6.34</td>
<td>6.31</td>
<td>6.33</td>
<td>6.29</td>
</tr>
<tr>
<td>trace-nyield-soonactive-once-nosync-O2-min50-thrd3200-4</td>
<td>11.77</td>
<td>11.73</td>
<td>11.76</td>
<td>11.77</td>
<td>11.79</td>
</tr>
<tr>
<td>trace-nyield-soonactive-once-nosync-min50-thrd3200-4</td>
<td>9.68</td>
<td>6.33</td>
<td>9.73</td>
<td>6.32</td>
<td>6.33</td>
</tr>
<tr>
<td>trace-soonactive-nosync-min50-thrd3200-4</td>
<td>7.64</td>
<td>10.26</td>
<td>10.58</td>
<td>8.01</td>
<td>10.38</td>
</tr>
<tr>
<td>optimizing-1</td>
<td>9.72</td>
<td>9.92</td>
<td>9.85</td>
<td>9.77</td>
<td>10.18</td>
</tr>
<tr>
<td>optimizing-once-1</td>
<td>10.25</td>
<td>10.26</td>
<td>10.61</td>
<td>10.10</td>
<td>10.17</td>
</tr>
<tr>
<td>optimizing-once-O1-1</td>
<td>10.33</td>
<td>10.27</td>
<td>10.36</td>
<td>10.36</td>
<td>10.24</td>
</tr>
<tr>
<td>optimizing-once-O2-1</td>
<td>20.54</td>
<td>20.51</td>
<td>20.66</td>
<td>20.32</td>
<td>20.41</td>
</tr>
<tr>
<td>tcs-adaptive-O2-noout-1</td>
<td>15.89</td>
<td>15.76</td>
<td>15.92</td>
<td>14.06</td>
<td>16.09</td>
</tr>
<tr>
<td>tcs-adaptive-noout-1</td>
<td>7.70</td>
<td>8.15</td>
<td>8.54</td>
<td>8.03</td>
<td>7.54</td>
</tr>
<tr>
<td>tcs-baseline-noout-1</td>
<td>214.59</td>
<td>214.29</td>
<td>214.39</td>
<td>214.43</td>
<td>214.40</td>
</tr>
<tr>
<td>tcs-baseline-tasksout-1</td>
<td>145.30</td>
<td>145.44</td>
<td>145.35</td>
<td>145.26</td>
<td>145.30</td>
</tr>
<tr>
<td>tcs-baseline-tracesout-1</td>
<td>144.23</td>
<td>144.45</td>
<td>144.28</td>
<td>144.32</td>
<td>144.22</td>
</tr>
<tr>
<td>tcs2-adaptive-1</td>
<td>7.54</td>
<td>7.55</td>
<td>7.76</td>
<td>7.91</td>
<td>7.82</td>
</tr>
<tr>
<td>tcs2-adaptive-O2-1</td>
<td>11.34</td>
<td>10.76</td>
<td>11.10</td>
<td>11.29</td>
<td>12.67</td>
</tr>
<tr>
<td>tcs2-adaptive-backward-1</td>
<td>8.35</td>
<td>8.15</td>
<td>8.05</td>
<td>8.41</td>
<td>8.30</td>
</tr>
<tr>
<td>tcs2-baseline-1</td>
<td>151.41</td>
<td>151.47</td>
<td>151.75</td>
<td>151.40</td>
<td>151.41</td>
</tr>
<tr>
<td>tcs2-baseline-backward-1</td>
<td>101.18</td>
<td>100.90</td>
<td>100.95</td>
<td>100.92</td>
<td>100.96</td>
</tr>
</tbody>
</table>

Table C.5: Wallclock times for JGF MolDynBenchSizeA.
<table>
<thead>
<tr>
<th>Configuration</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline-1</td>
<td>54.80</td>
<td>54.80</td>
<td>54.92</td>
<td>54.90</td>
<td>54.85</td>
</tr>
<tr>
<td>baseline-counters-1</td>
<td>56.25</td>
<td>56.27</td>
<td>56.27</td>
<td>56.25</td>
<td>56.35</td>
</tr>
<tr>
<td>counters-1</td>
<td>33.51</td>
<td>33.56</td>
<td>34.14</td>
<td>33.34</td>
<td>34.30</td>
</tr>
<tr>
<td>default-1</td>
<td>32.90</td>
<td>33.02</td>
<td>30.83</td>
<td>29.71</td>
<td>32.47</td>
</tr>
<tr>
<td>default-time-1</td>
<td>29.41</td>
<td>31.30</td>
<td>31.03</td>
<td>31.04</td>
<td>34.12</td>
</tr>
<tr>
<td>trace-noyield-nosync-min50-thrd3200-4</td>
<td>11.23</td>
<td>11.02</td>
<td>10.96</td>
<td>11.11</td>
<td>11.11</td>
</tr>
<tr>
<td>trace-noyield-soonactive-min50-thrd3200-4</td>
<td>11.25</td>
<td>11.24</td>
<td>10.98</td>
<td>11.24</td>
<td>10.95</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-4</td>
<td>11.07</td>
<td>11.01</td>
<td>11.04</td>
<td>11.26</td>
<td>11.07</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-4</td>
<td>31.83</td>
<td>30.85</td>
<td>31.11</td>
<td>32.57</td>
<td>31.49</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-2</td>
<td>17.60</td>
<td>17.20</td>
<td>18.33</td>
<td>17.41</td>
<td>18.42</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-3</td>
<td>12.70</td>
<td>12.72</td>
<td>12.92</td>
<td>12.82</td>
<td>12.39</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-4</td>
<td>11.17</td>
<td>11.10</td>
<td>11.22</td>
<td>11.08</td>
<td>11.02</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min100-thrd3200-4</td>
<td>11.06</td>
<td>11.07</td>
<td>11.08</td>
<td>11.33</td>
<td>10.97</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-4</td>
<td>11.14</td>
<td>11.00</td>
<td>11.25</td>
<td>11.02</td>
<td>11.02</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-1</td>
<td>31.04</td>
<td>31.41</td>
<td>31.88</td>
<td>31.49</td>
<td>32.38</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-2</td>
<td>18.11</td>
<td>17.98</td>
<td>17.95</td>
<td>18.73</td>
<td>18.41</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-3</td>
<td>12.37</td>
<td>12.49</td>
<td>12.56</td>
<td>12.57</td>
<td>12.47</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-4</td>
<td>10.98</td>
<td>11.01</td>
<td>11.17</td>
<td>10.94</td>
<td>11.22</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-thrd3200-4</td>
<td>10.92</td>
<td>12.21</td>
<td>11.01</td>
<td>11.04</td>
<td>10.89</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-1</td>
<td>31.60</td>
<td>31.30</td>
<td>31.13</td>
<td>31.22</td>
<td>32.15</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-2</td>
<td>17.98</td>
<td>17.37</td>
<td>18.02</td>
<td>18.24</td>
<td>18.34</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-3</td>
<td>13.21</td>
<td>12.97</td>
<td>12.65</td>
<td>12.70</td>
<td>13.21</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-4</td>
<td>11.40</td>
<td>11.36</td>
<td>11.36</td>
<td>11.70</td>
<td>11.45</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-1</td>
<td>30.88</td>
<td>31.01</td>
<td>31.38</td>
<td>31.48</td>
<td>32.05</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-2</td>
<td>17.37</td>
<td>17.59</td>
<td>17.44</td>
<td>17.31</td>
<td>18.14</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-3</td>
<td>12.95</td>
<td>12.54</td>
<td>12.69</td>
<td>12.48</td>
<td>12.88</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-4</td>
<td>11.04</td>
<td>11.05</td>
<td>11.26</td>
<td>11.04</td>
<td>11.23</td>
</tr>
<tr>
<td>trace-noyield-soonactive-once-nosync-O1-min50-thrd3200-4</td>
<td>11.38</td>
<td>11.05</td>
<td>12.05</td>
<td>12.01</td>
<td>11.18</td>
</tr>
<tr>
<td>trace-noyield-soonactive-once-nosync-O2-min50-thrd3200-4</td>
<td>11.51</td>
<td>11.91</td>
<td>11.39</td>
<td>11.72</td>
<td>11.44</td>
</tr>
<tr>
<td>trace-noyield-soonactive-once-nosync-min50-thrd3200-4</td>
<td>11.27</td>
<td>11.13</td>
<td>11.11</td>
<td>12.09</td>
<td>11.19</td>
</tr>
<tr>
<td>trace-soonactive-nosync-min50-thrd3200-4</td>
<td>11.14</td>
<td>10.92</td>
<td>9.98</td>
<td>11.03</td>
<td>11.04</td>
</tr>
<tr>
<td>optimizing-1</td>
<td>30.86</td>
<td>32.50</td>
<td>31.80</td>
<td>32.03</td>
<td>31.74</td>
</tr>
<tr>
<td>optimizing-once-1</td>
<td>32.74</td>
<td>32.82</td>
<td>32.82</td>
<td>32.75</td>
<td>32.74</td>
</tr>
<tr>
<td>optimizing-once-O1-1</td>
<td>32.81</td>
<td>32.83</td>
<td>32.80</td>
<td>32.78</td>
<td>32.81</td>
</tr>
<tr>
<td>optimizing-once-O2-1</td>
<td>39.02</td>
<td>39.02</td>
<td>38.95</td>
<td>38.92</td>
<td>38.93</td>
</tr>
<tr>
<td>tcs-adaptive-O2-noout-1</td>
<td>43.26</td>
<td>43.85</td>
<td>41.65</td>
<td>43.64</td>
<td>43.00</td>
</tr>
<tr>
<td>tcs-adaptive-noout-1</td>
<td>43.91</td>
<td>43.49</td>
<td>42.88</td>
<td>43.56</td>
<td>41.85</td>
</tr>
<tr>
<td>tcs-baseline-noout-1</td>
<td>69.61</td>
<td>69.68</td>
<td>69.64</td>
<td>69.63</td>
<td>69.56</td>
</tr>
<tr>
<td>tcs-baseline-tasksout-1</td>
<td>64.40</td>
<td>64.44</td>
<td>64.31</td>
<td>64.25</td>
<td>64.28</td>
</tr>
<tr>
<td>tcs-baseline-tracesout-1</td>
<td>60.40</td>
<td>60.41</td>
<td>60.38</td>
<td>60.41</td>
<td>60.42</td>
</tr>
<tr>
<td>tcs2-adaptive-1</td>
<td>42.09</td>
<td>43.30</td>
<td>43.49</td>
<td>41.44</td>
<td>41.99</td>
</tr>
<tr>
<td>tcs2-adaptive-O2-1</td>
<td>42.43</td>
<td>41.71</td>
<td>43.25</td>
<td>41.50</td>
<td>42.66</td>
</tr>
<tr>
<td>tcs2-adaptive-backward-1</td>
<td>43.02</td>
<td>40.73</td>
<td>42.99</td>
<td>41.66</td>
<td>41.53</td>
</tr>
<tr>
<td>tcs2-baseline-1</td>
<td>58.20</td>
<td>58.33</td>
<td>58.17</td>
<td>58.20</td>
<td>58.31</td>
</tr>
<tr>
<td>tcs2-baseline-backward-1</td>
<td>56.92</td>
<td>56.85</td>
<td>56.84</td>
<td>56.94</td>
<td>56.76</td>
</tr>
</tbody>
</table>

Table C.6: Wallclock times for JGFMonteCarloBenchSizeA.
<table>
<thead>
<tr>
<th>Configuration</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline-1</td>
<td>102.11</td>
<td>102.07</td>
<td>102.03</td>
<td>102.07</td>
<td>102.03</td>
</tr>
<tr>
<td>baseline-counters-1</td>
<td>101.03</td>
<td>101.03</td>
<td>101.06</td>
<td>101.04</td>
<td></td>
</tr>
<tr>
<td>counters-1</td>
<td>27.68</td>
<td>27.80</td>
<td>27.70</td>
<td>27.56</td>
<td>27.74</td>
</tr>
<tr>
<td>default-1</td>
<td>28.56</td>
<td>28.46</td>
<td>28.47</td>
<td>28.47</td>
<td>28.44</td>
</tr>
<tr>
<td>default-time-1</td>
<td>28.40</td>
<td>28.28</td>
<td>28.10</td>
<td>28.28</td>
<td>28.16</td>
</tr>
<tr>
<td>trace-noyield-nosync-min50-thrd3200-4</td>
<td>15.52</td>
<td>15.55</td>
<td>15.64</td>
<td>15.87</td>
<td>15.87</td>
</tr>
<tr>
<td>trace-noyield-soonactive-min50-thrd3200-4</td>
<td>20.20</td>
<td>20.78</td>
<td>20.71</td>
<td>20.39</td>
<td>20.56</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-4</td>
<td>18.23</td>
<td>17.62</td>
<td>18.11</td>
<td>17.90</td>
<td>17.63</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-1</td>
<td>28.12</td>
<td>27.58</td>
<td>27.43</td>
<td>28.14</td>
<td>27.47</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-2</td>
<td>18.67</td>
<td>18.62</td>
<td>18.60</td>
<td>18.62</td>
<td>18.60</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-fourlevels-min50-thrd3200-4</td>
<td>15.86</td>
<td>16.00</td>
<td>15.42</td>
<td>15.70</td>
<td>15.96</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min100-thrd3200-4</td>
<td>26.75</td>
<td>26.92</td>
<td>26.92</td>
<td>26.77</td>
<td>26.82</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-4</td>
<td>17.68</td>
<td>18.45</td>
<td>18.04</td>
<td>18.07</td>
<td>17.85</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-1</td>
<td>27.61</td>
<td>26.99</td>
<td>27.03</td>
<td>26.95</td>
<td>26.94</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3202-2</td>
<td>18.60</td>
<td>18.71</td>
<td>18.72</td>
<td>18.60</td>
<td>18.60</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-3</td>
<td>16.33</td>
<td>16.20</td>
<td>16.35</td>
<td>16.28</td>
<td>16.59</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-min50-thrd3200-4</td>
<td>15.83</td>
<td>15.67</td>
<td>15.76</td>
<td>15.48</td>
<td>15.72</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-thrd3200-4</td>
<td>16.04</td>
<td>15.74</td>
<td>15.92</td>
<td>15.84</td>
<td>16.19</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-1</td>
<td>27.98</td>
<td>27.91</td>
<td>27.98</td>
<td>28.00</td>
<td>27.87</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-2</td>
<td>18.96</td>
<td>18.97</td>
<td>19.07</td>
<td>19.02</td>
<td>19.09</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-time-min50-thrd3200-4</td>
<td>16.26</td>
<td>16.35</td>
<td>16.11</td>
<td>16.31</td>
<td>16.15</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-1</td>
<td>28.07</td>
<td>27.60</td>
<td>28.03</td>
<td>27.43</td>
<td>28.07</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-2</td>
<td>18.67</td>
<td>18.54</td>
<td>19.03</td>
<td>18.51</td>
<td>18.59</td>
</tr>
<tr>
<td>trace-noyield-soonactive-nosync-twolevels-min50-thrd3200-4</td>
<td>15.47</td>
<td>15.65</td>
<td>15.86</td>
<td>15.38</td>
<td>15.77</td>
</tr>
<tr>
<td>trace-noyield-soonactive-once-nosync-O1-min50-thrd3200-4</td>
<td>15.37</td>
<td>15.44</td>
<td>15.41</td>
<td>15.87</td>
<td>15.45</td>
</tr>
<tr>
<td>trace-noyield-soonactive-once-nosync-O2-min50-thrd3200-4</td>
<td>18.30</td>
<td>18.25</td>
<td>18.27</td>
<td>18.26</td>
<td>18.36</td>
</tr>
<tr>
<td>trace-noyield-soonactive-once-nosync-min50-thrd3200-4</td>
<td>15.34</td>
<td>15.81</td>
<td>15.34</td>
<td>15.37</td>
<td>15.38</td>
</tr>
<tr>
<td>trace-soonactive-nosync-min50-thrd3200-4</td>
<td>16.32</td>
<td>16.04</td>
<td>15.79</td>
<td>15.98</td>
<td>15.68</td>
</tr>
<tr>
<td>optimizing-1</td>
<td>27.48</td>
<td>27.35</td>
<td>27.35</td>
<td>27.36</td>
<td>27.43</td>
</tr>
<tr>
<td>optimizing-once-1</td>
<td>27.29</td>
<td>27.27</td>
<td>27.31</td>
<td>27.28</td>
<td>27.28</td>
</tr>
<tr>
<td>optimizing-once-O1-1</td>
<td>27.30</td>
<td>27.28</td>
<td>27.31</td>
<td>27.28</td>
<td>27.30</td>
</tr>
<tr>
<td>tcs-adaptive-O2-noout-1</td>
<td>33.75</td>
<td>32.19</td>
<td>35.11</td>
<td>38.20</td>
<td>36.21</td>
</tr>
<tr>
<td>tcs-adaptive-noout-1</td>
<td>32.19</td>
<td>31.17</td>
<td>31.95</td>
<td>31.48</td>
<td>32.33</td>
</tr>
<tr>
<td>tcs-baseline-noout-1</td>
<td>116.86</td>
<td>116.80</td>
<td>116.74</td>
<td>116.79</td>
<td>116.74</td>
</tr>
<tr>
<td>tcs-baseline-tasksout-1</td>
<td>116.58</td>
<td>116.52</td>
<td>116.50</td>
<td>116.52</td>
<td>116.47</td>
</tr>
<tr>
<td>tcs2-adaptive-1</td>
<td>27.94</td>
<td>27.93</td>
<td>27.90</td>
<td>28.13</td>
<td>27.93</td>
</tr>
<tr>
<td>tcs2-adaptive-backward-1</td>
<td>28.28</td>
<td>28.11</td>
<td>28.32</td>
<td>28.43</td>
<td>28.07</td>
</tr>
<tr>
<td>tcs2-baseline-1</td>
<td>101.87</td>
<td>101.81</td>
<td>101.83</td>
<td>101.84</td>
<td>101.81</td>
</tr>
<tr>
<td>tcs2-baseline-backward-1</td>
<td>103.06</td>
<td>103.08</td>
<td>103.01</td>
<td>103.03</td>
<td>103.05</td>
</tr>
</tbody>
</table>

Table C.7: Wallclock times for JGFUeulerBenchSizeA.
Figure C.3: Different Jikes RVM configurations.
Bibliography


---

4Workshop does not have page numbers.

5Workshop does not have page numbers.


\(^6\)Workshop does not have page numbers.
[26] D. Dice et al. Applications of the adaptive transactional memory test platform. In
Proc. of Workshop on Transactional Computing\(^7\), 2008.


of sequential programs. In Proc. of Conference on Programming Language Design


[30] Ics 161: Design and analysis of algorithms lecture notes for february 20, 1996:
-eppstein/161/960220.html.


optimizations for trace cache microprocessors. In Proc. of International Symposium

May 2005.

[34] B. Goetz. Optimistic thread concurrency: Breaking the scale barrier. Azul Whitepa-


\(^7\)Workshop does not have page numbers.


---

*8Workshop does not have page numbers.*


---

9Workshop does not have page numbers.


¹⁰Workshop does not have page numbers.


\(^\text{11}\)Workshop does not have page numbers.