Measuring and Improving the Potential Parallelism of Sequential Java Programs

Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Master of Science in the
Graduate School of The Ohio State University

By

Kevin Van Valkenburgh
Graduate Program in Computer Science and Engineering

The Ohio State University

2009

Thesis Committee:

Atanas Rountev, Advisor

P. Sadayappan

Abstract

There is a growing need for parallel algorithms and their implementations, due to the continued rise in the use of multi-core machines. When trying to parallelize a sequential program, software engineers are faced with the problem of understanding and removing inhibitions to parallelism in the context of high-level program constructs that are used to implement modern software. This thesis describes a dynamic analysis to analyze potential parallelism in Java software, based on coarse-grain parallelism among method calls. We find that this automated analysis, when applied to 26 sequential Java programs, provides characterization and insights about potential parallelism bottlenecks. We also perform case studies on several of the analyzed programs, locating specific elements in the implementation that impede potential parallelism, and altering these elements to find greater potential parallelism. Our studies indicate that the dynamic analysis could provide valuable feedback to a software engineer in support of program understanding and transformations for parallelism.

Acknowledgments

I would like to thank my advisor, Dr. Atanas Rountev, for his availability, interesting ideas, mentoring, and valuable feedback. His clarity of focus in identifying the important elements of a problem and separating them from the ancillary details made for an interesting and satisfying research experience. Thank you, Professor, for giving me an opportunity to work on such an interesting project.

Additionally, I would like to thank my former supervisor, Dr. Brian Robinson, for inspiring me to seek a graduate education. Before working for him as an undergraduate co-op, I was unaware of the depth of knowledge that existed in my field beyond a Bachelor's Degree.

Finally, I would like to thank my parents, who have always taken an interest in my academic and professional pursuits. Their positive feedback has always helped me in working towards and valuing my own achievements.

Vita

9 Jul, 1984 .... Born - Pennsylvania

Aug 2002 – May 2007 .... Bachelor of Science
Computer Science
Case Western Reserve University
Cleveland, Ohio

Jan 2005 – Aug 2005 .... Process Improvement Co-op
ABB Incorporated
Wickliffe, Ohio

May 2006 – Dec 2006 .... Process Improvement Co-op
ABB Incorporated
Wickliffe, Ohio

Fields Of Study

Major Field: Computer Science and Engineering

Contents

List of Figures

List of Tables

Chapter 1: Introduction

There has been a great deal of recent work in creating parallel algorithms. In the latter half of the past decade, it has become increasingly important to construct parallel software in not only the domains of scientific and high-performance computing, but in desktop software as well. Inspired by the transition of the hardware industry towards multi-core systems, this trend will not subside, and will likely only become more pressing as the need for software that can take advantage of multi-core hardware in more and more problem domains increases.

The classic challenges of designing parallel software include the difficulty of thinking in terms of a parallel solution to a sequential problem and in comprehending the scope of interactions of separate algorithms that make up a single solution. Further, given an existing, sequential program, it is often not apparent what design changes could be made to create a parallel version of that same program. A substantial number of previous works have introduced automated methods to measure potential parallelism in existing algorithm implementations [1,4,6,10]. The focus of these works has been to define a model of parallelism – that is, assumptions about what code can execute simultaneously with other code, and rules for under what circumstances an instruction would be executed – and then provide a means of describing the discovered potential parallelism of existing programs, perhaps through a number or plot describing the execution.

Much of this prior work investigates potential parallelism at a low level; that is, it explores the relationships of individual instructions or statements during a program execution and examines how they would be executed in the work's model of parallelism. Other work chooses specific control structures in a program implementation, such as loops [6], and focuses on analyzing potential parallelism at that level. Although it is very useful to understand potential parallelism at a low level for the sake of optimizations, there is more room to explore parallelism at a higher level.

It is the goal of the work in this thesis to explore potential parallelism at a level that corresponds to the parallelism tools that exist in the Java programming language today, such as threads, and in this way provide means of describing potential parallelism that would be readily available to a software designer using Java. The starting point of our work is an automated tool, created by Prof. Rountev, for analyzing sequential Java programs and characterizing their potential parallelism. The tool uses a method-based model of parallelism that corresponds with the options of how a software designer would choose to divide work among threads. This approach represents an optimistic view of what speedup could be achieved by a parallel version of the same program, redesigned at a high level to take advantage of the implementation's potential parallelism.

The first contribution of this thesis is an extensive experimental study of applying the tool to 26 sequential Java benchmarks from four well-known benchmark suites: SPEC JVM98, Java Grande, Olden (Java version), and DaCapo. This study provides initial insights about the amount of potential parallelism in these benchmarks. Our second contribution is an extension of the analysis which characterizes the contribution of different sources of data dependences on the potential parallelism. When

applied to the 26 benchmarks from the first study, the extended analysis reveals which categories of memory locations are the most likely inhibitors of parallelism. The third contribution of this thesis is several case studies of Java programs, in which the analysis tool is used to guide semantics-preserving transformations of these programs and to evaluate their effect on their potential parallelism.

The remainder of this thesis is organized as follows. In Chapter 2, we review the problems that give rise to this work, two models of parallelism, and a conceptual method of measuring parallelism in sequential Java programs. In Chapter 3, we introduce an implementation of this potential parallelism measurement and present measurements of 26 sequential Java benchmarks. In Chapter 4, we look at several benchmarks more closely and perform alterations to reveal greater theoretical speedup by identifying and removing impediments to parallelism in their implementation, while respecting their original semantics. Related work is discussed in Chapter 5. Finally, Chapter 6 summarizes the results and contributions of this work as well as possible directions of future study.

Chapter 2: Background

In this chapter we discuss a dynamic analysis of potential parallelism. This analysis, applied to Java programs, presents a means of describing the potential parallelism in a single-threaded Java application. Although the analysis applies to Java programs, it is fundamentally inspired by Kumar's analysis of potential parallelism in FORTRAN programs [4]. The analysis algorithm itself was designed and implemented by Prof. Rountev. Starting with this analysis, we have (1) performed an experimental study of applying the analysis to sequential Java benchmarks, (2) designed and implemented analysis extensions to characterize the effects of various sources of data dependences on the potential parallelism, and (3) performed case studies in which the analysis is used to guide and evaluate semantics-preserving transformations for increasing potential parallelism. For the subsequent discussion, it is expected that the reader be familiar with program instrumentation techniques in general.

In summary, the analysis augments an existing, single-threaded Java application, such that its original semantics is preserved, but that it also gathers data about dependences between program statements. The result of the augmentation is that the program, in addition to its original output, generates data describing the ideal potential parallel processing for the original, unaltered program. This augmentation is implemented by instrumenting the original program such that extra code is added to track and record relevant events such statement executions, method invocations, and the uses (both reads and writes) of distinct program variables, fields, and array

elements. From the data gathered, determinations can be made as to whether there are dependences between instructions. The output of the analysis is a speedup score, which, roughly speaking, describes the ratio of the logical time it takes for the original program's code to execute, to the logical time that it would take for that program to execute in an ideal environment in which all potential parallelism of the code could be exploited. (The concepts of *logical time*, *ideal execution environment*, and *potential parallelism* will be described in detail later in this section.) There are both static and dynamic aspects of this analysis, including the instrumentation of the program and the gathering of data from its execution.

2.1 Goal of the Analysis

The goal of this analysis is to create objective, repeatable means of measuring characteristics of Java programs. Measuring characteristics such as data dependences (through local variables, static and instance fields, and array elements) can then provide some insight as to the structure and behavior of a program and how its implementation's execution time could be affected by dependences. The analysis measures data to reflect the code's inherent ability to have as many statements be executed as early as is possible; this is to say that, based on some models of *ideal execution environment* and a *model of parallelism*, the analysis determines the earliest time that each statement can be executed based on its control and data dependences on the statements that would be executed before it in a sequential-execution environment.

2.2   Models of Parallelism

To be able to analyze a program in a way such that we can describe characteristics such as dependences between program statements and the potential parallelism inherent in its implementation, we must first review what dependences are and what it means to think of a sequential Java program in a parallel manner.

2.2.1   Dependences

Dependences between statements are the restrictions that cause the execution time of those statements to be decided by one statement preparing data for, or allowing the execution of, the other. The analysis of potential parallelism determines the time that a statement can execute by taking control dependences and data dependences into account and finding the earliest logical time at which the statement can be executed with respect to said dependences, and with respect to the model of parallelism as will be described in Section 2.2.4. How these dependences are determined is described below.

Control Dependences and Potential Parallelism

In its most basic form, control dependence represents a relationship of two program statements in which the execution of the former statement determines that the latter will be executed. A formal definition of control dependence can be found in [3]. Control dependences are typically introduced into a program by conditional structures such as *if* or *while.* The importance of control dependences in evaluating the potential parallelism in Java programs is that, no matter how many independent portions of a program are to be executed in parallel, if there is a control dependence of statement

B on statement A, statement B's execution will be delayed to after the time at which statement A has executed.

Data Dependences and Potential Parallelism

Every program statement reads from some number of memory locations and writes to at most one memory location. In this document, we will use a representation of Java code in which a typical statement can read from at most two memory locations and write to at most one. Speaking informally and generally, this intermediate representation used in the Soot analysis framework [11], called Jimple, represents complex Java statements that contain multiple expressions as multiple statements that contain each a single expression, but retain exactly the semantics of the original statements. More formally, the details of this language are presented in the Jimple grammar given in Appendix A.

In a strictly sequential-execution environment, whenever a statement is executed, the memory locations from which it is reading are "ready" by the nature of sequential execution, which is to say that the values at these input locations have already been calculated in preparation for this statement to be executed. Likewise, the memory location that the statement writes to (if any) in a sequential execution is in a state such that the statement can write to it and there are no concerns regarding program correctness over the value that will be placed in this location.

However, in an environment in which parallel execution is possible, statements that access the same memory locations become data dependent on one another. To preserve the semantics of a sequential execution of a program, the parallel version's statements cannot be executed until the memory locations that it accesses are in an

analogous state to the one that they would have been in during the sequential execution of the program. This gives rise to the three classic types of data dependences, which we will review below.

Flow Dependences

In a flow dependence, statement B reads from some memory location that was written to by a prior-executed statement A. Statement B must wait until after statement A's write is complete to perform its read.

An example of a flow dependence exists in Figure 2.1, between the statements at lines 3 and 4. Since statement 3 writes to the local variable b, and statement 4 reads from that location, statement 4 is flow-dependent on statement 3.

```
3:   b = 6;
4:   c = a + b;
```

Anti-Dependences

In an anti-dependence, statement B writes to a location from which a value was read by a prior-executed statement A. Statement B must wait until after statement A's read is complete to perform its write.

An example of an anti-dependence exists in Figure 2.1, between the statements at lines 5 and 6. Since statement 6 writes to the local variable b, and statement 5 reads from that location, statement 6 is anti-dependent on statement 5.

```
5:   d = a + b;
6:   b = 7;
```

Output Dependences

In an output dependence, statement B writes to some location that was written to by a prior-executed statement A. Statement B must wait until after statement A's write is complete to perform its own write.

An example of an output dependence exists in Figure 2.1, between lines 4 and 8. Since statement 4 writes to the local variable `c`, and statement 8 also writes to that location, statement 8 is output-dependent on statement 4.

```
4:  c = a + b;
...
8:  c = a + e;
```

2.2.2   An Abstract Execution Environment

Let us now define the ideal execution environment and model of parallelism used by the analysis, so that we may speak more firmly as to how the analysis makes decisions about dependences and potential speedup. As in Kumar's work, we assume the existence of an ideal machine on which we can execute our program under test. This hypothetical machine has several key properties:

1. It has as many processors as necessary to run any number of simultaneous program statements

2. It can manage simultaneous statement executions in such a way that there is no overhead in joining these separate "processes"

3. It can manage memory in such a way that there is no overhead in sharing memory

The fact that no such machine can exist is irrelevant, as we are not trying to automate some process that would speed up a program, but instead we are using this ideal machine as architecture-independent means of measuring program characteristics.

To lead into the explanation of the model of parallelism that we are using, let us first describe the concepts of speedup and logical time. The speedup could be thought of as the ratio of an original program's execution in real-world physical time to the improved physical execution time of a parallel version of that program. However, it is unsuitable to define speedup in this way for the purposes of our analysis for

two reasons. First, it is impractical because the analysis does not yield a parallel version of the program to run. Second, such a measurement would be dependent on the machine on which it is executed and the resource usage of other tasks that the machine is performing at the time of execution. Instead, we define a unit of logical time to be the amount of time that it takes for a single statement to be executed. We can then describe speedup as the ratio of the number of statements executed in an unaltered execution of the program, to the logical time at which the last statement would be executed in a parallel version of the program in which control and data dependences are respected. We can describe the execution time of any particular statement with a *timestamp*, that logical time at which it is executed.

Note that a compound statement can be made up of multiple expressions, perhaps including expressions such as some combination of method invocations and arithmetic operations. For the purpose of determining the logical time of execution for a compound statement, we consider it to be broken apart into separate statements that correspond to each of its component expressions, and instead of assigning a single logical execution time to the compound statement, we assign logical execution times to the simpler intermediate statements. This separation will be discussed further in Section 2.4.3.

Again, this abstraction of time is meant to provide a means of reasoning about time that is independent of a physical machine. A desired consequence of considering all statements to execute in a single unit of time is that a statement that performs simple addition between two operands would take the same single unit of logical time as a multiplication of those operands. Although a physical machine would typically

execute such instructions in different amounts of physical time, such hardware-specific differences can be ignored here.

2.2.3 Fine-Grain Parallelism

We can now further describe our model of parallelism. There can be multiple approaches for defining a model of parallelism with different purposes, so we will discuss two separate approaches here.

The first model, as used in Kumar's work [4], is a fine-grain approach. In this approach, statements are considered to be the smallest building blocks of parallel execution. This is to say that the program can be thought of as a series of statements, and when the next statement is going to be scheduled, it can be scheduled at the earliest time at which its dependences have been satisfied, with no other restrictions.

Based on these definitions of dependences, we can now further describe statement scheduling in the fine-grain model. Given the restrictions that we have discussed, the ideal machine, which is aware of what statements will execute during a run of a program, is free to schedule all instructions at the earliest logical time at which their dependences are satisfied. A statement that would be among the last to execute in a long sequential execution of the program, but that has no dependences on statements that would be executed earlier in that sequential execution, could be scheduled to execute at a very early logical time in the fine-grain model.

An application of this model could be in considering instruction-level parallelism on the scale of one or several methods in a program. Note that in this fine-grain approach, two in-order program statements with no data dependences could be scheduled at the same time. Later in this chapter, we will discuss a different model in

```
1:  int a, b, c, d, e, f;
2:  a = 5;
3:  b = 6;
4:  c = a + b;
5:  d = a + b;
6:  b = 7;
7:  e = 8;
8:  c = a + e;
9:  d = b + e;
10: f = 9;
```

Figure 2.1: Example code for intraprocedural fine-grain parallelism.

which the execution of the second may be required to wait for the execution of the first, thus introducing program structure as a factor in scheduling.

Fine-Grain Examples

Let us explore our model of fine-grain parallelism through viewing example code and describing the logical execution times of the statements therein. Although the example code is contrived, it serves to explain the relationships of both read and write accesses to memory locations.

Suppose we have the block of Java code given in Figure 2.1. The execution times of the statements can be found in Table 2.1. In this example, we see that in the fine-grain model, a statement that would be executed late in a sequential execution of the code, such as statement 10, can be executed early in a parallel execution. We also see that statements with dependences on statements that have other dependences, and so on, are the statements that are executed later in a parallel execution.

Let us now consider an example that crosses the boundaries of two methods. Suppose we begin with `Method1` being called in the Java code given in Figure 2.2.

Stmt	Time	Explanation
2	1	No dependences, can execute immediately
3	1	No dependences, can execute immediately
4	2	Flow dependence on a and b, written at time 1 by statement 1 and 2
5	2	Same as statement 4
6	3	Anti-dependence on b, read at time 2 by statement 4 and 5
7	1	No dependences
8	3	Output dependence on c, written at time 2 by statement 4 Flow dependence on a, written at time 1 by statement 2 Flow dependence on e, written at time 1 by statement 7
9	4	Flow dependence on b, written at time 3 by statement 6 Output dependence on d, written at time 2 by statement 5 Flow dependence on e, written at time 1 by statement 7
10	1	No dependences

Table 2.1: Timestamps for intraprocedural fine-grain parallelism.

```
1:   void Method1(X m1x) {        7:   void Method2(X m2x) {
2:       m1x.a = 1;               8:       int z = m2x.a;
3:       Method2(m1x);            9:       z = z + 1;
4:       m1x.b = 2;               10:      m2x.a = z;
5:       int y = m1x.a;           11:      m2x.c = z; }
6:       int w = m1x.b; }
```

Figure 2.2: Example code for interprocedural fine-grain parallelism.

The execution times for the statements are given in Table 2.2. Again in this example we see that a statement at the end of the execution, at line 6, can have an earlier timestamp if it does not depend on the execution of any statement that has a later timestamp. Additionally, we see that some statements such as the one on line 5 can have a dependence on a statement from another method (the one on line 10) and as such have its execution delayed until after the statement from the other method is executed.

13

Stmt	Time	Explanation
1	1	Method invoked
2	2	Flow dependence on m1x from method invocation
3	2	Flow dependence on m1x from method invocation
4	2	Flow dependence on m1x from method invocation
5	6	Flow dependence on m2x.a from statement 10 Flow dependence on m1x from method invocation
6	3	Flow dependence on m1x.b from statement 4 Flow dependence on m1x from method invocation
7	2	Method invoked
8	3	Flow dependence on m1x.a from statement 2 Flow dependence on m2x from method invocation
9	4	Flow and output dependence on z from statement 8
10	5	Flow dependence on z from statement 9 Anti-dependence on m2x.a from statement 8 Output dependence on m1x.a from statement 2 Flow dependence on m2x from method invocation
11	5	Flow dependence on z from statement 9 Flow dependence on m2x from method invocation

Table 2.2: Timestamps for interprocedural fine-grain parallelism.

2.2.4   Method-Based Coarse-Grain Parallelism

A second model is a course-grained approach. In general, instead of treating individual program statements as the atomic units that are run in parallel, a higher-level structure such as a method is treated as the portion of the program that can be run in parallel with other portions of the program. The analysis uses this method-based coarse-grain parallelism.

In the method-based model, we can still have an arbitrarily large number of statements executing at one time in our ideal execution environment, just as in the fine-grain approach. In contrast to the fine-grain approach, in the method-based model, all statements inside a single method invocation are assumed to execute sequentially.

This means that, in addition to control dependences determined by control statements, there is an implied dependence of each of the method's in-order program statements on the previous one, such that the latter statement cannot execute until the former has completed. This, of course, means that control dependences within a method are satisfied in a parallel execution by respecting them just as they would be in a sequential execution. In terms of the ideal execution environment, this could be thought of as granting a method exactly one of the infinite processors on which to run, and all of the statements in the method must execute sequentially on that processor. As such, this model is less optimistic than the fine-grain one, in which there is no such limitation on statements inside one method.

Where the parallelism comes into play in this model is when methods are called, regardless of whether they are static / non-static, or void / non-void. Once a method M_2 is invoked, the statements in that new method invocation can be executed in parallel to the statements in the calling method M_1; the execution time of the first statement in the callee is the unit of time just after that of the statement in M_1 which called it. In terms of the ideal execution environment, this process can be thought of as a second of the ideal machine's processors being made available exclusively to execute the statements in the callee M_2, again in a sequential manner inside that method. Statements from both the caller and the callee method can be executed at the same time; in this way, we allow parallelism based on the program design structure of methods. Further parallelism can be exploited by invoking more methods at the same time; if the original caller M_1 calls a different method M_3, or another instance of method M_2, then all of these methods can be executing statements in parallel. If

callees such as M_2 or M_3 call other methods as well, then all of these methods can be executing statements at the same time.

Although the method-based model enjoys the potential for statements from separate method invocations to be executed at the same logical times, the model does limit the potential parallelism in certain ways. As in the fine-grain model, control and data dependences can delay a statement's execution. Note that these limitations through dependences mean that although the statements in a single method invocation are executed in sequence, it is not necessarily the case that any statement's execution time will be one unit of time after the statement in that method whose execution preceded it. Also a consequence of respecting data dependences is that, when we have a method M_1 calling another non-void method M_2, and M_1 assigns the return value of M_2 to a some variable, there is a dependence introduced between the assignment operation in M_1 and the computational result of M_2. Although the assignment of the return value is scheduled to take place at the logical time at which the method call to M_2 returns, statements in M_1 following the call to M_2 are free to be scheduled to execute in parallel with it, provided that they are not data dependent on its return value.

An application of this model could be designing software at the method level to take advantage of multi-threading.

Coarse-Grain Examples

Let us explore coarse-grain parallelism through a pair of examples. Suppose we have the block of Java code given in Figure 2.2, which is again entered via a call to Method1; this means we have the same code and the same calling context as in

the previous fine-grain parallelism example, but we are instead using the coarse-grain model.

Stmt	Time	Explanation
1	1	Method invoked
2	2	Sequential control dependence on method invocation
		Flow dependence on m1x from method invocation
3	3	Sequential control dependence on prev. statement
		Flow dependence on m1x from method invocation
4	4	Sequential control dependence on prev. statement
		Flow dependence on m1x from method invocation
5	7	Sequential control dependence on prev. statement
		Flow dependence on m2x.a from statement 10
		Flow dependence on m1x from method invocation
6	8	Sequential control dependence on prev. statement
		Flow dependence on m1x.b from statement 4
		Flow dependence on m1x from method invocation
7	3	Method invoked
8	4	Sequential control dependence on method invocation
		Flow dependence on m1x.a from statement 2
		Flow dependence on m2x from method invocation
9	5	Sequential control dependence on prev. statement
		Flow and output dependence on z from statement 8
10	6	Sequential control dependence on prev. statement
		Flow dependence on z from statement 9
		Anti-dependence on m2x.a from statement 8
		Output dependence on m1x.a from statement 2
		Flow dependence on m2x from method invocation
11	7	Sequential control dependence on prev. statement
		Flow dependence on z from statement 9
		Flow dependence on m2x from method invocation

Table 2.3: Timestamps for coarse-grain parallelism.

The execution times of the statements are given in Table 2.3. The execution times of statements 3 and 4 increase relative to the prior example (Fine-Grain Parallelism, Example 2; the timetable thereof is presented in Table 2.2) because of sequential

control dependence on prior-executed statements from `Method1`; in the fine-grain model, statements 2, 3, and 4 could all have been executed at the same time.

The execution times of the statements in `Method2` must now be delayed relative to the prior example because they are dependent on the execution time of statement 3, which is later in this example. Consequently, statement 8's execution timestamp has been increased by one. Next, although statements 9 and 10 are control dependent on their predecessor statements, their execution times only raise by one over that of their predecessor statements, just as in the prior example, because the introduced control dependences are dependences on the same statements as their data dependences are. Finally, the execution time of statement 11 goes up, because although it is flow-dependent on statement 9, it is also control dependent on statement 10, so it must be executed after statement 10.

Returning to the scope of `Method1`, statement 5's execution time increases by one from the prior example simply because it is flow-dependent on statement 10, which has a greater execution time in this example. Although statement 5 is control dependent on statement 4, the flow dependence supersedes the control dependence, because the dependence dictating the latest execution time is the one that is considered. Finally, statement 6 must execute at the given later time now, because of its control dependence on statement 5.

Let us also consider another coarse-grain example with more interprocedural dependences, with the code given in Figure 2.3, in which the statements have timestamps as given in Table 2.4. Note that this example diverges from the others in that we call method `Method2` multiple times. This means that some statements are executed

```
1:  void Method1(X m1x) {              9:      void Method2(X m2x) {
2:      m1x.a = 1;                     10:         m2x.b = m2x.a;
3:      m1x.c = 2;                     11:         int n = m2x.c;
4:      Method2(m1x);                  12:         int p = 3;
5:      m1x.a = 3;                     13:         m2x.b = n; }
6:      Method2(m1x);
7:      int y = Method3(m1x);          14:     int Method3(X m3x) {
8:      m1x.b = y; }                   15:         m3x.a = 1;
                                       16:         return m3x.a; }
```

Figure 2.3: Second example code for coarse-grain parallelism.

twice; for those that are, the statement is listed twice in Table 2.4, with its line number and which particular execution it is, listed in the leftmost column. Thus this example explores dependences through memory locations accessed multiple times by statements in separate invocations of the same method. For the sake of removing noise from Table 2.4, only a subset of dependences are listed.

Observe that any invocation of Method2 writes to m2x.b. It is thus the case that two invocations of Method2 that are attempted to be executed concurrently will have their potential parallelism limited by output dependences through m2x.b. Note that the second invocation of Method2 must actually wait until the first has finished to proceed, based on the writes to m2x.b at the end of the first invocation and the beginning of the second invocation.

Also note that here we see a method Method3 returning a value that is stored in a local variable of Method1. While the void method call at line 4 was able to execute at time 4 and the following statement at line 5 was able to be executed immediately afterward, such timely control flow is not possible here at the analogous line pair of 7 and 8. This is because the non-void method call and subsequent assignment to a

19

Stmt and Invocation	Time	Explanation
1	1	Method invoked
2	2	Sequential control dependence on method invocation
3	3	Sequential control dependence on prev. statement
4	4	Sequential control dependence on prev. statement
5	6	Anti-dependence on m2x.a from statement 10/1st
6	7	Sequential control dependence on prev. statement
7	11	Flow dependence on return value of Method3
8	13	Output dependence on statement 13/2nd
9, 1st	4	Method invoked
10, 1st	5	Flow dependence on m1x.a from statement 2
11, 1st	6	Flow dependence on m1x.c from statement 3
12, 1st	7	Sequential control dependence on prev. statement
13, 1st	8	Output dependence on m2x.b from 10/1st
9, 2nd	7	Method invoked
10, 2nd	9	Output dependence on m2x.b from statement 13/1st
11, 2nd	10	Flow dependence on m1x.c from statement 3
12, 2nd	11	Sequential control dependence on prev. statement
13, 2nd	12	Output dependence on m2x.b from 10/2nd
14	8	Method invoked
15	10	Anti-dependence on m2x.a from statement 10/2nd
16	11	Flow dependence on m2x.a from statement 15

Table 2.4: Timestamps for the example from Figure 2.3.

local variable on line 7 requires the return value of `Method3` to complete. As such, the time for the completion of the statement on line 7 is the same as the time that the call to `Method3` returns. The statement on line 8 can be executed after this time.

2.3   An Overview of the Dynamic Analysis

The analysis of potential parallelism has been designed and implemented by Prof. Rountev in Java, using the Soot framework's instrumentation capabilities. The implementation of the analysis tool consists of two distinct components: a dynamic analysis of the program that collects data regarding dependences and potential speedup in the Java program under test, and a static analysis and instrumentation of that program. In this section, we will discuss the dynamic portion of the analysis.

After the program under test has been augmented with the instrumentation described in Section 2.4, we can execute the instrumented version to gather data about its potential parallel properties. The means by which the analysis gathers this data is the Tracker, a class that is loaded alongside the program under test, and to which calls are made from the instrumented program under test.

First, let us describe an addition to the original program that allows us to record data from the instrumented program. The program under test is manually modified to add a wrapper around its main() method. This wrapper method is very simple; it just includes a call to activate the Tracker and initialize its recording features, then a call to the application's original main() method, and then finally a call to shut down the Tracker and report the data that it has collected.

As has been mentioned above regarding the instrumentation, calls have been inserted into the program code so that data regarding method calls, statement executions, and the like are passed to the Tracker for storage. In this way, the data-recording is encapsulated in a new component, separate from the instrumented program under test.

2.3.1   Finding Dependences

We can find dependences by adding extra variables to the program under test and using them to describe characteristics of that program, such as when memory locations were accessed or when a statement was executed.

Finding Data Dependences

The analysis tracks data dependences via shadow variables, as inspired by Kumar's work in [4]. A shadow variable is an extra variable, of which there are two varieties (*write* shadows and *read* shadows), that is inserted into the augmented program. The analysis makes use of both read shadow variables and write shadow variables to keep track of the respective access times at which program variables have been read and written. An original variable may have one of both types of shadows, and any one shadow only expresses a read or write access time for one specific original variable. In general, all class fields, both static and instance, have read and write shadows, as do all array elements. Also, the analysis considers dependences through local variables, but only when those local variables are assigned the return value of a method call. The reason that dependences through local variables that take return values from methods are considered is that, to be consistent with the method-based coarse-grain model, the use of a return value from a method must be restricted to take place after

22

the completion of that method. Thus, a statement A that is placed sequentially inside a method Y after a statement that calls some other method X should not be able to execute until after the call to X is completed, if A contains the return value of X as a local variable in the set of values that it reads. This manner of tracking dependences through local variables, combined with the method's sequential statement execution within a single method, requires only write shadows for local variables, as will be described further in Section 2.4.3.[1]

The analysis uses these shadow variables to decide the earliest time at which a statement could be executed. Consider that an arbitrary statement may read zero or more input memory locations, and write to zero or one output memory locations. The statement's input memory locations make up its *read set*, whereas the statement's output locations make up its *write set*. From the inputs and outputs in these sets, it is straightforward to determine the proper shadow variables that are relevant in finding data dependences; there is a one-to-one mapping from an arbitrary program variable/field/array element to its read or write shadow. For any operation, the execution time of the statement can be no earlier than the timestamp value contained in any of the shadow variables for memory locations in the read set or the write set, plus one. As mentioned earlier, the addition of one represents the fact that the memory location to which the shadow corresponds has been accessed at the time given in the shadow, and at least one unit of logical time must pass before this subsequent access to it may be performed.

---

[1]Soot's Jimple representation does not use the formal parameters of a method inside the method body; the formals are simply assigned to locals at the very beginning of the method. Thus, it is not necessary to have shadows for formal parameters.

Finding Control Dependences

The analysis considers control dependence as follows. Inside a method, each time a statement is executed, the analysis increases the value in a `control` shadow variable. This control variable is a single extra variable added to the scope of a method that is used to track control dependences for each statement in that method. The execution time of a statement can be no less than the `control` value when that statement is executed, plus one. (Data dependences can increase this value further, as described above.) The addition of one to the `control` value is a result of the fact our model represents the sequential execution of two subsequent statements in a method, and at least one unit of logical time must pass from one statement's execution to the next. In a world of completely sequential statement execution and no data dependences, `control` would be analogous to a program counter; in a world with some degree of parallel statement execution and with data dependences delaying statement executions, `control` can be thought of as a descriptor specifying the earliest logical time at which a statement can be executed. If data dependences do not delay the execution of the statement, then its timestamp will be decided strictly by the `control` value, and its timestamp will be set to `control` + 1, and `control` will be set to `control` + 1. If data dependences are a factor in the statement's execution time, then `control` can be set to a larger value than `control` + 1.

Let us now explain how control dependence is observed across method boundaries. When a method is invoked, it is passed the `control` value of the parent method at the call site — that is, the value of `control` when the statement calling the new method is executed. This means that the first statement of the called method and the subsequent statement in the parent method can potentially be executed at the

same logical time, with the caveat that data dependences may influence the execution time of either one of those statements. Thus it can be seen that this concept is the core of the method-based parallelism model; once a method is called, its statements can execute at the same logical time as statements from other methods.

Using the logic described for control and data dependences, we have explained how the analysis determines the time at which a statement can execute. From this point, it is straightforward to explain how the values for read and write shadows are calculated. Initially, both types of shadows can be set to zero, since no operations have been performed on them and there have been no dependences detected yet. Then, for a given statement execution, a read or write shadow variable for a memory location accessed by the statement will be updated by assigning that statement's execution timestamp value to it.

2.3.2   Shadow Definitions Example

Here we will revisit our example of coarse-grain parallelism, and include the definitions of the shadow variables for the local variables and fields used in the code. Write shadows will be prefixed with `ws_`, and read shadows will be prefixed with `rs_`. The special `control` shadow variable will simply be named `control`.

Class instance fields have shadows that are defined as fields of the object. For each method, `control` is a local variable, initialized with the value of `control` from its caller at the site of its invocation. Local variables have locally-defined shadows, on the condition that they take a return value from a method; otherwise, they have no shadows (as is the case for `y`, `z`, and `w` from the example).

```
1:   class X {                          12: void Method2(X m2x) {
2:       public int a, b, c;            13:    int control = ... // caller's value
3:       public int ws_a, ws_b, ws_c;   14:    int z = m2x.a;
4:       public int rs_a, rs_b, rs_c; } 15:    z = z + 1;
                                        16:    m2x.a = z;
5:   void Method1(X m1x) {              17:    m2x.c = z; }
6:       int control = ... // Initialize to caller's control value
7:       m1x.a = 1;
8:       Method2(m1x);
9:       m1x.b = 2;
10:      int y = m1x.a;
11:      int w = m1x.b; }
```

Figure 2.4: Coarse-grain example with shadow declarations.

The instrumentation that manages the values in these shadow variables will be presented in Section 2.4.4.

2.4 Static Analysis and Instrumentation

The static analysis is performed on the program under test to add shadow variables and the method invocations that are performed during a later dynamic analysis of that program. There are three main categories of instrumentation processes performed on the program, which can best be explained by describing them individually.

2.4.1 Class-Level Instrumentation

The behavior of the static analysis on the class level is fairly straightforward: for each class, its fields will be given both read shadow variables and write shadow variables, which are new fields of that class with the same access levels. This means that all reads and writes to any class field can be tracked by the dynamic analysis. The shadows of instance fields are instance fields themselves (e.g., as in Figure 2.4); the shadows of static fields are, of course, static fields.

```
int m(objType1 a, ojbType2 b)
{
    // Local declarations
    // Some statements
    return retVal; // there may be several of these, not necessarily one
}
```

Figure 2.5: A general, uninstrumented method m().

2.4.2   Method-Level Instrumentation

On the method level, there are a number of changes made by the instrumentation to prepare the target program for the dynamic analysis. These changes include:

1. Create shadow variables for local variables

2. Add administrative variable `control`

In Steps (1) and (2), the analysis adds more locally-scoped variables to the method. In Step (1), the analysis creates write shadow variables for the original local variables. Note that the only locals that need shadows are those that receive return values from method calls, so it is possible that no locals will be given corresponding shadows in this step. Further, these locals only require write shadows, as will be discussed shortly in Section 2.4.3. In Step (2) the analysis adds a `control` shadow variable that helps to track statement execution time by respecting the sequential program's statement execution order, as described in 2.2.4.

This instrumentation can be summarized by the generalized examples in Figure 2.5, an uninstrumented method, and Figure 2.6, an instrumented method. *ComputeStatementTime* is a method of the dynamic analysis that determines the logical time at which a statement can be executed; it will be defined in Section 2.4.3.

27

```
int m(objType1 a, ojbType2 b)
{
        // Original local declarations
        int c;        // an example
        int retVal;  // contains the return value
        // Extra administrative local
        int control;
        // Declare shadow variables for locals
        // that take method return values
        int ws_c;
        int ws_retVal;
        control = DynamicAnalysis.GetLastTimeStamp();
        // Original statements, plus instrumentation
        DynamicAnalysis.ComputeStatementTime(control, ws_retVal);
        return retVal;
}
```

Figure 2.6: The general, instrumented method m().

2.4.3   Statement-Level Instrumentation

There can be many different types of statements in an arbitrary Java program, from simple assignments of literal values to local variables, to assignment of parenthesized compound mathematical expressions to class instance fields, to method invocations that take an arbitrary number of parameters, and so on. The analysis must take all of these types of statements into account when performing instrumentation, such that it can generate timestamps properly for each of them. Handling such a great variety of statement types could be a fairly complicated task, but fortunately Soot provides us with a simple means of addressing the entirety of this variety such that we can perform all of the functions described in the base analysis design, while changing none of the semantics of the original program.

Soot represents the Java bytecode as statements of certain forms; there are few enough of these forms that it is manageable for the analysis to explicitly take each one

28

into account. As mentioned previously, the Jimple grammar in Section 6 describes the composition of a Java statement in terms of Jimple statements, the intermediate Java representation language used by Soot.

The instrumentation of a statement adds the necessary bookkeeping code to calculate the statement's execution time, based on the data that it reads and writes, and on program control. We will use the following auxiliary functions to help describe the instrumentation of the original program's statements. Let $ComputeStatementTime$ be defined as the function

$$ComputeStatementTime(control, read\_set, write\_set) \\ = Max(control, MaxSet(read\_set), MaxSet(write\_set)) + 1$$

where $Max(...)$ takes any number of integral parameters and returns the maximum of all of these integers. Similarly, $MaxSet(...)$ takes a set of integers and returns the maximum of all of these integers, or 0 if the set is empty. Let us also use the helper function $GetLastTimeStamp$. Every call to $ComputeStatementTime$ has the side-effect of setting a persistent `last_calculated_timestamp` integer variable to that function's return value. Let $GetLastTimeStamp$ merely return this value.

Performing the Instrumentation

Of the possible Jimple statements, the below types are of interest to the static analysis. In the descriptions of the statements' instrumentation, let the read shadow of some variable named `data` be denoted as `rs_data`. Likewise, let the write shadow of some variable named `data` be denoted as `ws_data`.

Static Field Read

Statement Type

A read of a static field

Form

```
local := static_field
```

How it is instrumented

```
1:  control := ComputeStatementTime(control, {}, {ws_static_field})
2:  rs_static_field := Max(control, rs_static_field)
3:  <original statement>
```

There are two considerations that will decide the earliest execution time of this statement. The first is the `control` value at the time the statement is executed, and the second is the write shadow for the static field. The timestamp for this statement is thus the maximum of these values, plus one. Instrumentation is inserted to calculate this maximum and assign it to the control variable, which represents when this statement can be executed. If the static field's read shadow is of lesser value than this statement's timestamp, then the read shadow is updated to be this timestamp. Since this operation is a read, the read shadow for the field does not play a role in delaying the statement's execution.

Let us also discuss the contribution of the local to this statement's instrumentation. Recall that locals will have write shadows (and not read shadows) if and only if they take the return value of a method call. Since this statement represents a write to a local, if it has a write shadow, one may think that is is necessary to update that shadow to the just-computed new value of `control`. However, such an update is unnecessary since all subsequent reads-from/writes-to this local variable will be assigned larger timestamps, due to the sequential nature of intraprocedural control flow (which is achieved with the use of `control` at each timestamp computation).

Static Field Write

Statement Type

A write of a static field

Form
```
static_field := local or
static_field := constant
```

How it is instrumented

```
1:  control := ComputeStatementTime(control,
    {rs_static_field}, {ws_static_field, ws_local})
2:  ws_static_field := control
3:  <original statement>
```

The static field cannot be written until after the last write and the last read to the field. Additionally, if the RHS of the assignment is a local variable, the static field cannot be written until after that local was written. The write shadow for the field will be set to the execution time of this statement, to reflect that the field was written at this time, and that subsequent accesses should not be allowed to take place until this new value has been written into the static field. It is possible that there will be no `ws_local` if the RHS of the assignment is a constant, or if the RHS is a local variable that is not given a write shadow.

Let us also discuss the contribution of the local to this statement's instrumentation. Recall that locals will have write shadows (and not read shadows) if and only if they take the return value of a method call. Since this statement represents a read from a local, the statement has to "wait" until the value of the local becomes available (i.e., until the method that provides this value returns to the caller).

Instance Field Read

Statement Type

A read of a field of a class instance

Form

31

```
local1 := local2.field
```

How it is instrumented

```
1:  control := ComputeStatementTime(control, {}, {ws_local2, local2.ws_field})
2:  local2.rs_field := Max(control, local2.rs_field)
3:  <original statement>
```

The instance field cannot be read until after the last write to the field, as provided by the ws_field write shadow field of the object pointed-to by local2. Additionally, the local2 pointer must have been set to the proper object before the read, so the read must take place after the assignment of the appropriate object reference to local2. It is possible that local2 does not have a shadow, in which case only the field's write shadow (i.e., local2.ws_field) is taken into account.

Instance Field Write

Statement Type

A write of a field of a class instance

Form
```
local1.field := local2 or
local1.field := constant
```

How it is instrumented

```
1:  control := ComputeStatementTime(control, {local1.rs_field},
    {ws_local1, ws_local2, local1.ws_field})
2:  local1.ws_field := control
3:  <original statement>
```

The local field cannot be written until after the last write and the last read to the field. Additionally, if the RHS of the assignment is a local variable, the instance field cannot be written until after that local was written. The same applies to the local variable used in the LHS. The write shadow for the field will be set to the execution time of this statement, to reflect that the field was written at this time, and that

subsequent accesses should not be allowed to take place until this new value has been

written into the local field. (As before, it is possible that there will be no `ws_local1`

and/or `ws_local2`.)

Array Element Read

Statement Type

A read of an element in an array

Form
`z := x[y]`

In this form of statement, `z` is a local, `x` is a local pointing to an array on the
heap, and `y` is either a constant or a local. Let `rs_<x[y]>` and `ws_<x[y]>` refer to the
read and write shadows of `x[y]`, respectively.

How it is instrumented

```
1:  control := ComputeStatementTime(control, {}, {ws_x, ws_y, ws_<x[y]>})
2:  rs_<x[y]> := Max(control, rs_<x[y]>)
3:  <original statement>
```

The array element cannot be read until after the last write to that element. Addi-

tionally, the array pointer `x` must have been set to the proper array before the read,

and the index value stored in `y` must have been set as well. (As before, it is possible

that there will be no shadows for `ws_x` and/or `ws_y`.)

Array Element Write

Statement Type

A write to an element in an array

Form
`x[y] := z`

How it is instrumented

```
1:  control := ComputeStatementTime(control,
```

```
    {rs_<x[y]>}, {ws_x, ws_y, ws_z, ws_<x[y]>})
2:  ws_<x[y]> := control
3:  <original statement>
```

The array element cannot be written until after the last write and the last read to that element. Additionally, the write shadows of x, y, and z should be taken into account (if they exist). The write shadow for the array element will be set to the execution time of this statement, to reflect that the element was written at this time, and that subsequent accesses should not be allowed to take place until this new value has been written into the array element.

Method Call

Statement Type

An invocation of a method, either void or non-void

Form
myMethod() or
myMethod(param1, ...) including this as a parameter, or
returnVal = myMethod(...) or
returnVal = myMethod(param1, ...) including this as a parameter

How it is instrumented

A method call is instrumented differently depending on whether there is an assignment to returnVal, as will be explained below.

Assuming that there is no assignment to a returnVal, the instrumentation would be the following:

```
1:  control := ComputeStatementTime(control, {}, {ws_param1, ...})
2:  <original statement>
```

Assuming that there is an assignment to a returnVal, the instrumentation would instead be:

```
1:   control := ComputeStatementTime(control, {}, {ws_param1, ...})
2:   <original statement>
3:   ws_returnVal = GetLastTimeStamp()
```

The method invocation requires a unit of logical time to complete, so the `control`

value is increased before the call, to the maximum of the `control` value and the

parameters' write shadows, plus one. This time simply represents the earliest time

at which all of the parameter values are available, and at which program control can

reach the statement. In the case that we have a return value assignment, the write

shadow of that return value is set to the last calculated timestamp, which represents

the time at which the method invocation completes, and thus the time that this return

value becomes available.

Note that the coarse-grain model we have proposed assumes that all calls are

non-blocking; that is, all method invocations can run in parallel with their callers.

It would be possible to consider some calls to be blocking — preventing the further

execution of statements in the calling method until the called method returns — and

the instrumentation that would be performed on a blocking void method call would

be as follows.

```
1:   control := ComputeStatementTime(control, {}, {ws_param1, ...})
2:   <original statement>
3:   control := GetLastTimeStamp()
// And finally, if and only if there is a return value assignment
// in the <original statement>...
4:   ws_returnVal := control
```

The difference is that here, in Statement 3, we need to make certain that the

`control` value after the call is equal to the value of the last timestamp calculated in

the called method, which enforces the serial nature of the blocking call by pushing

back the execution time of subsequent statements from the caller to after this last

timestamp.

Void or Non-Void Return

Statement Type

A return statement in a method, either of the *void* type or some non-*void* type

Form
```
return or
return local or
return const
```

How it is instrumented
```
1:  ComputeStatementTime(control, {}, {ws_local})
2:  <original statement>
```

Given our non-blocking model for parallel method calls, the effect this code has is that *ComputeStatementTime* updates the `last_calculated_timestamp` to the timestamp of the return, so that the calling method can obtain it through *GetLastTimeStamp*. That is, this statement's instrumentation enables the instrumentation for a method call to assign the proper timestamp to the write shadow of a local that takes a method call's return value, because it sets the `last_calculated_timestamp` to the time at which the method call has completed.

Identity Statement

Statement Type

A statement added as part of the Jimple representation for parameter passing.

Form
```
local := parameter
```

How it is instrumented

We instrument all of the consecutive identity statements in a method together. There can be zero or more identity statements at the beginning of a method, after the declaration of local variables.

36

```
1:    <first original statement>
2:    <second original statement>
      ... // n total identity statements
n+1:  control := GetLastTimeStamp()
```

Identity statements are Jimple statements in which a value is assigned from a
formal parameter (including the implicit parameter `this`) to a local variable. They
appear at the very beginning of the method body. We do not increment `control`, and
instead just assign to it the latest timestamp that had already been calculated (at
the call site). Note that the instrumentation that initializes `control` will be inserted
at the start of the method even if there are no identity statements there. An identity
statement can also occur at the beginning of a `catch` block to represent the caught
exception; such occurrences are handled similarly.

Other Statements

Statement Type

Any other statement type

Form
This may include types of statements such as
`local1 := local2` or
`local := local2 + local3`, or
`local := const`, and others.

How it is instrumented

```
1:  control := ComputeStatementTime(control, {}, {ws_RHS_1, ws_RHS_2})
2:  <original statement>
```

Statements other than the ones described before this section are of minimal im-
portance to the analysis, and it suffices to record their execution times by observing
in-method sequential control flow and the write shadow **ws_RHS** of any input operand

that may appear in the statement. (By the way that Soot decomposes Java statements into Jimple, there will be at most two input operands in this statement.) It can be the case that there is no write shadow for the right-hand side, such as in the case that it is a constant, and then this write shadow drops out of the calculation.

2.4.4   Calculating Shadows Example

Here we present an example of instrumented code as described in Section 2.4.3 by building on the example from Section 2.3.2. That code demonstrated where the shadows are initially declared; Figure 2.7 adds the instrumentation to calculate shadows for each statement. For simplicity of presentation we use the formal parameters `m1x` and `m2x` in the code even though, as discussed earlier, in reality the Jimple representation uses local variables in their place (initialized by identity statements).

Note that the calls to *ComputeStatementTime* immediately before the return statements appear to be unnecessary, since for them there do not exist matching calls to *GetLastTimeStamp* at the corresponding call sites. However, consider the example where a method M_1 calls a non-void *native* method M_2, which in turn calls a void method M_3. Both M_1 and M_3 will be instrumented, but of course M_2 cannot be instrumented. The call site inside M_1 that calls M_2 will need to obtain the last timestamp, and this timestamp may have to be established at the time when M_3 returns to M_2, by the instrumentation of the return statement in M_3.

```
1:  class X {
2:      public int a, b, c;
3:      public int ws_a, ws_b, ws_c;
4:      public int rs_a, rs_b, rs_c;
5:  X() {
6:      ws_a = ws_b = ws_c = 0;       // this is only for illustration;
7:      rs_a = rs_b = rs_c = 0; } }  // fields are initialized to 0 by the JVM

8:  void Method1(X m1x) {
9:      int control = GetLastTimeStamp();
10:     control = ComputeStatementTime(control, {m1x.rs_a}, {m1x.ws_a});
11:     m1x.ws_a = control;
12:     m1x.a = 1;
13:     control = ComputeStatementTime(control, {}, {});
14:     Method2(m1x);
15:     control = ComputeStatementTime(control, {m1x.rs_b}, {m1x.ws_b});
16:     m1x.ws_b = control;
17:     m1x.b = 2;
18:     control = ComputeStatementTime(control, {}, {m1x.ws_a});
19:     m1x.rs_a = Max(control, m1x.rs_a);
20:     int y = m1x.a;
21:     control = ComputeStatementTime(control, {}, {m1x.ws_b});
22:     m1x.rs_b = Max(control, m1x.rs_b);
23:     int w = m1x.b;
24:     ComputeStatementTime(control, {}, {});
25:     return;
}

26: void Method2(X m2x) {
27:     int control = GetLastTimeStamp();
28:     control = ComputeStatementTime(control, {}, {m2x.ws_a});
29:     m2x.rs_a = Max(control, m2x.rs_a);
30:     int z = m2x.a;
31:     control = ComputeStatementTime(control, {}, {});
32:     z = z + 1;
33:     control = ComputeStatementTime(control, {m2x.rs_a}, {m2x.ws_a});
34:     m2x.ws_a = control;
35:     m2x.a = z;
36:     control = ComputeStatementTime(control, {m2x.rs_c}, {m2x.ws_c});
37:     m2x.ws_c = control;
38:     m2x.c = z;
39:     ComputeStatementTime(control, {}, {});
40:     return;
}
```

Figure 2.7: Course-grain example with instrumentation.

Chapter 3: Measurements and Characterization of Potential Parallelism

3.1 Initial Measurements

The analysis has been applied to a set of 26 Java benchmark programs to evaluate their potential parallelism, in the manner described in Chapter 2. The set of benchmarks analyzed were taken from SPEC JVM98, the Java Grande Sequential Benchmark set v2.0, a Java version of the Olden benchmarks, and the DaCapo 2006-MR2 Benchmark Suite.

3.1.1 Experimental Setup

Preparing the benchmark programs for analysis required the construction of a harness script and a slight modification to each of the benchmarks so that the harness script could instrument and execute them all in a uniform manner. For most benchmarks, it also required the choice of what input or problem size would be used for that program, which was typically chosen from a set of inputs provided with the benchmark release.

The Benchmarks Used In This Thesis

While many of the benchmarks provided in the above-mentioned suites were of value to this analysis, some were not, and so there had to be a distinction between benchmarks that were included in the experimental suite and those that were not.

Since the purpose of the analysis was to analyze sequential programs, it was the case that the analysis would not have any applicability to any multi-threaded programs in these suites, so those were not included in the experiments.

The SPEC JVM98 benchmark suite includes mostly single-threaded applications that are "real-world" programs or that featured an algorithm used frequently in the real world, and so it was relevant to include these benchmarks in the analysis. All of the SPEC JVM98 benchmarks were used in this thesis except `mtrt`, a multi-threaded raytracer, and `check`, a program with the simple purpose of checking implementation behaviors of the JVM. The single-threaded `raytrace` benchmark was used in the place of `mtrt`.

The Java Grande Sequential Benchmark Suite, version 2.0, includes three main "sections" of benchmarks: Low Level Operations, Kernels, and Large Scale Applications. The Large Scale Applications were chosen to be included in this thesis because they were most similar to higher-level software systems, the likes of which are intended to be the target of this analysis. These benchmarks were obtained from the Edinburgh Parallel Computer Centre's website.

The Java Olden benchmarks were all included, since each of the seven is sequential. Note that these benchmarks are a Java version of the Olden benchmarks, and that they were used in [8] and were obtained from the authors of that work.[2]

Finally, the DaCapo benchmarks included 11 programs, of which four are multi-threaded; all sequential programs were included in the analysis. It should be noted that at the time of this writing, although the DaCapo website indicates that only three of the benchmarks are multi-threaded, it is the case that the DaCapo `eclipse`

[2]We would like to thank Mark Marron for providing this benchmark code.

benchmark uses multi-threading as well, and thus it was not included in the analysis. This benchmark set was introduced in [2] and its release was obtained from the DaCapo Benchmark Suite website.

Note that both Java Grande and SPEC JVM98 include a raytracer benchmark. For the sake of clarity in this document, when the context does not indicate which raytracer is under discussion, we will distinguish between these by referring to them with the shorthand **Spec raytrace** and **JG raytracer**.

Alterations to the Benchmarks

Each benchmark required a set of minor adjustments to fit into the test environment. The chief modification to each was the inclusion of a new class called `WrapperMain`. This class consists of one `main()` method that invokes whatever appropriate method or combination of methods is necessary to execute the benchmark. In the case of most of the benchmarks, this was a very simple wrapper that invoked a `main()` method or a surrogate thereof in that program.

Many of the benchmarks relied on a test harness provided by their benchmark suite to be initialized with a proper configuration at the start of execution. Oftentimes the provided harness would call a benchmark setup method that would load or create any input data required by the benchmark, and in some other cases, the input data location was specified in a set of static fields. Typically, preparing a benchmark for execution under the analysis involved minor alterations to the benchmark's setup method or the main workload method to point to the proper input files and directories.

Input and Problem Sizes

The SPEC JVM98, Java Grande, and DaCapo benchmark suites contain a notion of separate *input sizes*, in which the benchmarks could be executed on different input data sets. In general, this meant that it was possible to measure speedup for a benchmark for input sizes that caused that program to perform different amounts of work. The SPEC JVM98 suite included three different input size choices, including 1, 10, and 100, with 1 as the smallest and 100 as the greatest, although these sizes were not strictly proportional to the integers 1, 10, and 100. The Java Grande Sequential Benchmark suite's Section 3 benchmarks included two input sizes A and B, with A as the smaller size and B as the larger size. The DaCapo benchmarks had a range of inputs from *Small* to *Default* to *Large*, sometimes even including an *Extra Small* input, with the *Default* and *Large* inputs occasionally being the same for some of the programs. In contrast, the Java Olden benchmarks only had one default input size provided.

For most of the benchmarks, it was found that one of the problem sizes that was released standard with the benchmark suite represented an appropriate amount of work for that benchmark to process, in measuring its speedup. In general, when possible, these input sizes were selected in such a manner so that a benchmark would not repeat the same work multiple times on the same input data. In cases where a larger input size simply meant that the benchmark was performing the same work multiple times, to no different end than just processing that set of data just once, typically a smaller input size that performed a set of work once would be selected. (An example of such repetitive work would be the SPEC JVM98 `compress` benchmark's problem size 100, which loops over the same input five times.)

43

We selected input sizes for the benchmarks based on the amount of work being performed on the input. For a small number of benchmarks, we found that there were enough statements being executed for the minimum problem size that the analysis was overflowing the `int` values used to represent the timestamps. Thus, we manually reduced the size of those programs' input to yield a shorter execution to measure. This limitation is purely artificial and will be removed in future versions of the analysis through the use of `long` timestamp values. The four relevant benchmarks are listed in Table 3.2.

Table 3.1 describes the nature of the benchmarks. The problem sizes used are listed in Table 3.3, and those that are denoted as having custom input are explained in Table 3.2.

In the case of one benchmark, DaCapo's `bloat`, it was found that the benchmark's input was its own compiled code. This presented a challenge to the analysis, in that instrumenting the `bloat` benchmark would effectively change the input on which it was running, thus making it difficult to compare the instrumented and uninstrumented executions of the program. The benchmark was instead provided a class from `JLex` 1.2.6, `JLex.CMinimize`, which was around the same size (in bytes) as the original small input `EDU.purdue.cs.bloat.trans.ValueNumbering`.

The Harness Script

The benchmarks suites were set up so that they could be instrumented and executed all from one Perl script. The script was placed in the root directory of the testing environment, and each benchmark was placed in a subdirectory at that location. Each benchmark directory contained the instrumented benchmark, as well as a configuration file called `config.txt`, which described the full name of its `WrapperMain`

Table 3.1: Benchmarks: nature of computation.

Benchmark	Nature
SPEC JVM98	
compress	An LZM file compression utility
db	A database using flat files
jack	A parser generator
javac	A Java compiler
jess	A rule-based expert system
mpegaudio	An MPEG-3 audio stream decoder
raytrace	A basic raytracer with shadows, reflection, refraction
Java Grande Sequential - Section 3	
euler	Computational fluid dynamics
moldyn	A molecular dynamics simulation
montecarlo	A Monte Carlo simulation of stock prices
raytracer	A basic raytracer with shadows, reflection, refraction
search	Alpha-beta pruned search of "Connect Four" game
Java Olden	
bisort	A bitonic sort of a set of N numbers
bh	An n-body Barnes-Hut computation
em3d	Models electromagnetic waves through objects in three dimensions
health	Simulates treatment of patients in Columbian health care system
power	Models a power network, sets prices to maximize economic efficiency of consumption
tsp	A randomized algorithm to solve the Traveling Salesman problem
voronoi	Generates a random set of points and creates a Voronoi diagram from them
DaCapo	
antlr	An LL parser generator
bloat	A Java bytecode optimization and analysis tool
chart	Plots PDF line graphs
fop	Parses XSL-FO and creates a corresponding PDF
jython	Interprets Python
luindex	Indexes a set of documents
pmd	Analyzes Java classes for source code problems

Table 3.2: Custom problem size adjustments.

Benchmark	Alteration
bloat	Replaced input bloat class file with JLex class file
em3d	Reduced: Nodes from 4000 to 2000, and Connections from 500 to 250
euler	Reduced interpolation scale factor from Size A of 8 to 4
moldyn	Reduced the data size from Size A of 8 to 6
JG raytracer	Reduced canvas size from Size A of 150x150 pixels to 75x75 pixels

class as well as any arguments it may require, including the size argument mentioned earlier in this section. The hand-altered version of the original benchmark was stored in a subdirectory called `original` off the benchmark's directory.

The harness script would then invoke Soot to process a benchmark using the instrumentation described in Section 2.4 and then execute a dynamic analysis of the benchmark to calculate speedup.

3.1.2   Initial Experiments With The Analysis

As mentioned previously, the chief output of the dynamic analysis is a speedup value, the ratio of logical time to execute the program in our coarse-grain parallel model respecting all dependences, to the logical time to execute the program sequentially. The computed speedups for the benchmarks are given in Tables 3.3. The analysis output differs slightly from run to run (on the same analyzed program), due to non-deterministic aspects of the execution introduced by the JVM — for example, the order of class loading and initialization through static initializer methods, and the finalizer invocations due to garbage collection. Our experiments show that the

variations in speedup values from run to run are neglibigle. The table shows the median speedup value out of three runs.

As can be expected, the instrumented versions of the benchmarks take additional time and memory to execute when compared to the benchmarks in their original, unaltered form. These running times and memory requirements are presented in Table 3.4. Finally, the number of classes and methods in each benchmark is also provided in Table 3.4 as a summary of the programs' relative sizes.

Execution durations were measured by running the program three times; the median value of the resulting run durations is shown in the table. Memory usage for the instrumented programs was measured by periodically calling the Java runtime's memory methods, and subtracting `Runtime.freeMemory()` from `Runtime.totalMemory()`. Memory usage for the uninstrumented, original benchmarks was determined by giving the benchmarks very lightweight instrumentation: each method was given an extra statement at the beginning that called into a dynamic analysis with the sole purpose of performing a memory check; this lightweight instrumentation has been treated as negligible for our memory measurement purposes. The memory measurements in the table are the median ones out of three runs.

Additionally, the numbers of application classes were determined by using Soot's `Scene.v().getApplicationClasses()` method at instrumentation time. All of the methods in these application classes were then counted as well, through each `SootClass`'s `getMethods()` method. Classes that were part of the dynamic analysis bookkeeping code were excluded, but the `WrapperMain` classes were included, since they now served as the main "driver" for each benchmark.

All experiments were performed on a Dell PowerEdge R300 with a Quad Core Intel® Xeon® X3363, 2.83GHz, 2x6M Cache, 1333MHz FSB and 8Gb of RAM. The execution environment was running a Linux 2.6.18 operating system. All experiments were executed on a Java HotSpot 64-Bit Server VM, build 1.6.0_11-b03. The Java libraries used were from this 1.6.0_11 release for Linux, and they were fully instrumented along with the benchmarks.

Table 3.3: Benchmarks: speedup and running time.

Benchmark	Size Chosen	Computed Speedup	Original Running Time (seconds)	Inst. Running Time (mm:ss)	Inst. : Original Slowdown
compress	10	1.2178	00.422	00:38.634	91.65
db	100	1.1634	05.522	01:50.456	20.00
jack	1	1.6865	00.615	00:06.349	10.33
javac	10	2.7790	00.755	00:05.730	7.59
jess	100	5.1821	01.478	02:23.981	97.41
mpegaudio	1	1.6375	00.501	00:35.925	71.69
Spec raytrace	100	2.4110	00.951	01:34.665	99.58
euler	custom	1.6424	01.415	15:24.547	653.17
moldyn	custom	1.8223	00.516	00:15.294	29.63
montecarlo	size A	1.2712	03.057	01:01.782	20.21
JG raytracer	custom	1.2211	00.574	00:25.270	44.00
search	size A	1.5775	02.071	08:20.895	241.83
bh	default	47.4109	01.539	05:42.893	222.81
bisort	default	1.1340	00.311	00:04.591	14.79
em3d	custom	3.9550	00.780	01:19.723	102.19
health	default	116.9337	01.491	00:29.075	19.49
power	default	134.1112	00.489	01:04.321	131.63
tsp	default	38.1510	01.793	00:25.456	14.20
voronoi	default	32.0687	01.814	01:22.891	45.70
antlr	small	1.6256	00.666	00:07.762	11.65
bloat	custom	5.1387	03.049	01:26.205	28.28
chart	small	2.0019	02.396	00:46.716	19.50
fop	default	2.3843	02.346	00:38.942	16.60
jython	small	2.2108	01.707	00:29.387	17.21
luindex	small	3.9669	00.950	00:28.103	29.57
pmd	small	2.0089	00.787	00:03.416	4.34

Table 3.4: Benchmarks: memory consumption and size of program.

Benchmark	Original Memory Usage (MB)	Inst. Memory Usage (MB)	Inst. : Original Memory Usage	Number of Classes	Number of Methods
compress	9.8	71.4	7.3	23	164
db	25.5	155.0	6.1	15	164
jack	5.6	51.5	9.2	68	445
javac	6.9	38.1	5.5	183	1300
jess	7.0	1107.1	157.5	161	804
mpegaudio	2.3	16.5	7.1	63	441
Spec raytrace	10.4	380.7	36.8	35	294
euler	35.1	20.6	0.6	5	31
moldyn	0.6	1.3	2.0	5	23
montecarlo	83.2	509.3	6.1	15	182
JG raytracer	1.2	3.7	3.2	13	73
search	11.0	2416.6	220.2	6	32
bh	14.6	2983.7	203.7	8	63
bisort	8.1	8.3	1.0	3	17
em3d	24.5	49.7	2.0	6	29
health	78.9	92.1	1.2	6	21
power	4.5	209.3	46.0	7	33
tsp	128.3	129.8	1.0	3	17
voronoi	154.5	763.2	4.9	7	67
antlr	4.1	44.1	10.8	229	2683
bloat	24.0	613.1	25.6	388	4260
chart	16.4	259.3	15.8	1145	13448
fop	20.2	171.9	8.5	3588	26135
jython	14.7	93.3	6.4	1497	13881
luindex	3.1	33.3	10.9	350	2740
pmd	3.3	19.3	5.9	1824	14391

3.1.3 Overhead

At this time, the analysis has been partially optimized for speed, but not for memory usage. The focus of the work has been on creating a functional version of the analysis to gather speedup information. For performance gains, the analysis uses some instrumentation optimization techniques, such as examining basic blocks and reducing sequences of certain types of instrumentation statements into one statement (an example is shown later). There is room to improve the memory overhead of the analysis, in that it currently stores persistent handles to array shadow variables and effectively prevents their garbage collection, and future work could improve upon this. It is possible that there are further opportunities to improve performance and memory usage, including but not limited to more sophisticated instrumentation techniques.

As can be seen, at this time there is a noticeable overhead for both the execution duration and the memory usage in instrumenting many of the benchmarks. For the set of benchmarks used in this thesis, the median slowdown (ratio of instrumented runtime to un-instrumented runtime) is 28.92, and 15 of the 26 benchmarks have a slowdown less than a factor of 30. The median memory overhead is a factor of 6.75, with 17 of the benchmarks showing a memory overhead of less than a factor of 10. Although this overhead is noticeable when compared to the original executions of the benchmarks, it is typical of instrumented programs to exhibit these kinds of costs. Further, it is not strictly necessary to run the analysis on very large inputs to programs, which would incur high execution durations and memory usage; it can be the focus of this analysis to find the potential parallelism through an execution of the "core" algorithms in a benchmark, in which case executing those algorithms a

limited number of times can reveal the same speedup as executing those algorithms many times.

In regards to the optimizations that have been included in the analysis, an example of their general form can be seen in Figure 3.1. In this code, the three original statements *S1*, *S2*, and *S3* are all sequentially-executing members of a basic block, and each operates on a read set and write set that have no shadow variables. In such a case, the calls to *ComputeStatementTime* can be removed and replaced with a call to *ComputeStatementTime_Enhanced*, which acts much in the same way that *ComputeStatementTime* does, but takes a second parameter n that represents the number of *ComputeStatementTime* calls that have been removed.

```
// Unoptimized Instrumentation
// All statements have no shadows for the read and write sets
1:  control := ComputeStatementTime(control, {}, {})
2:  <Original statement S1>
3:  control := ComputeStatementTime(control, {}, {})
4:  <Original statement S2>
5:  control := ComputeStatementTime(control, {}, {})
6:  <Original statement S3>

// Optimized Instrumentation
1: control := ComputeStatementTime_Enhanced(control, 3)
2: <Original statement S1>
3: <Original statement S2>
4: <Original statement S3>
```

Figure 3.1: Optimizing the instrumentation.

3.1.4 Discussion of Experimental Results

Our first observation is that the majority of these speedups, as determined by the analysis described in Chapter 2, are not remarkable: all of the benchmarks outside of the Java Olden suite have a speedup in the range from one to six. While a program's users would certainly appreciate a performance increase by a factor in that range, the speedup value by itself does not seem to indicate many opportunities for seriously improving the performance of those programs. Further, as was discussed in Chapter 2, the speedup value in our coarse-grained model of parallelism represents an ideal value that could not actually be achieved on general-purpose hardware; this means that if a programmer were to attempt to parallelize one of these benchmarks while keeping the implementation's inherent dependences intact, the resulting parallel program would have performance gains less than the speedup given in Table 3.3. To have a hope of finding greater performance gains, we will need to find a way to unearth greater speedup values from these programs, and this means that we will want to find a way to find the dependences that are limiting these speedup values.

In contrast, the Java Olden benchmarks appear to have a very high potential for speedup, with many in the double digits, and with the `power` and `health` benchmarks reporting speedups of over 100. This stark contrast suggests that there is a fundamental difference between implementation of the benchmarks in the Java Olden suite and those of the benchmarks in the other suites. Perhaps the Java Olden authors specifically programmed their implementations so that they would be readily parallelizable, as in fact the authors did write separate parallel versions of these programs, or perhaps the nature of these Java Olden computations is such that they are much more inherently parallelizable than other algorithms. Another explanation for this

high difference in speedups could be that the other benchmarks make use of utilities like file wrapper classes or other library data structures that impose additional parallelism-inhibiting dependences, and perhaps Java Olden does not do this.

Another observation is that the speedups of certain benchmarks do not necessarily relate to the inherent potential parallelism in the problems that they solve. For example, for the basic feature sets of the SPEC JVM98 and Java Grande raytracers, the author would expect to see a much higher speedup, since ray tracing is fairly easy to parallelize by hand, and the generic algorithm involves many non-dependent reads and non-dependent writes. We will explore the ray tracing problem further in Chapter 4 and find a way to reveal its hidden potential parallelism.

3.2 Contribution of the Java Libraries

In the above experiments, we have measured the potential parallelism of the benchmarks, including all of the code that is executed during a run of a program. Although these numbers are interesting and can be a starting point for future experiments, we can gather more data about the potential parallelism of a given program by recognizing that it is composed of two separate segments: user code and library code. We can separate the two by performing the same analysis as in Section 3.1, but using the original Java libraries instead of using their instrumented counterparts. The chief intent for such experiments would be to find the potential parallelism in the user code itself, which is under the control of the application designer, as opposed to that in the Java libraries, which is typically not under the control of an application designer. In examining the benchmarks in this way, while it can be expected that the analysis

would run more quickly if the library code is ignored, further questions arise, including what compromises this approach may bring to the correctness of the analysis.

3.2.1 Experiments With Instrumented Benchmarks and Uninstrumented Libraries

Table 3.5: Analysis with uninstrumented libraries.

Benchmark	Speedup: All Code Inst.	Speedup: Libraries Uninst.	Speedup Change %	Running Time Reduction %	Memory Usage Reduction %
compress	1.2178	1.2172	-0.1%	58.8%	1.4%
db	1.1634	1.0673	-8.3%	77.4%	25.4%
jack	1.6865	1.1720	-30.5%	78.6%	85.0%
javac	2.7790	2.7216	-2.1%	63.3%	77.0%
jess	5.1821	5.4902	5.9%	52.4%	14.9%
mpegaudio	1.6375	1.6299	-0.5%	87.3%	68.1%
raytrace	2.4110	2.4417	1.3%	46.8%	21.4%
euler	1.6424	1.6402	-0.1%	55.7%	1.8%
moldyn	1.8223	1.8221	0.0%	0.4%	48.2%
montecarlo	1.2712	1.5439	21.4%	47.8%	7.5%
raytracer	1.2211	1.2210	0.0%	11.1%	9.1%
search	1.5775	1.5775	0.0%	48.0%	17.0%
bh	47.4109	51.9712	9.6%	34.7%	21.0%
bisort	1.1340	1.1339	0.0%	3.1%	0.3%
em3d	3.9550	4.0588	2.6%	43.7%	0.0%
health	116.9337	47.1535	-59.7%	71.6%	16.5%
power	134.1112	134.8474	0.5%	60.4%	22.4%
tsp	38.1510	36.5441	-4.2%	6.7%	0.3%
voronoi	32.0687	32.1207	0.2%	32.2%	9.7%
antlr	1.6256	1.4013	-13.8%	55.7%	46.4%
bloat	5.1387	20.2789	294.6%	76.2%	77.7%
chart	2.0019	35.9533	1696.0%	77.5%	45.7%
fop	2.3843	1.9229	-19.4%	81.6%	79.0%
jython	2.2108	1.3617	-38.4%	56.2%	44.9%
luindex	3.9669	6.0863	53.4%	67.0%	62.5%
pmd	2.0089	4.3735	117.7%	56.9%	83.2%

As can be seen in Table 3.5, the savings in running time are consistently very large. The DaCapo benchmarks, which are in general the largest of the benchmarks

used in this thesis, show the greatest changes in speedup. These results indicate that the instrumentation of library code is necessary, since without it the speedup measurements could be quite different from the actual speedups that could be computed with instrumented libraries. Although the speedups listed in the table give us information about the potential parallelism in the benchmarks' user code, the results do not indicate any strong correlations in the relationships of speedups of user code versus speedups of entire programs.

3.2.2 Compromises to Accuracy

It is to be expected that the correctness of the analysis would be affected by running it on instrumented benchmarks with uninstrumented libraries, as compared to the original analysis, in which all of the code executed was being instrumented, and thus measured. The most obvious limitation is simply that dependences in the libraries cannot be measured, and so any data of contributions they have to inhibiting speedup would be lost; note that this can be a desirable affect if one intends to measure solely the contribution of user code to the program's speedup. However, it is important to consider any other effects that using uninstrumented code may have on the results of the analysis. Let us consider these effects here.

The following factors could influence correctness:

1. On the method level, is there some manner in which methods interact that relies on invalid assumptions when instrumented code calls into uninstrumented code?

2. Also on the method level, could there be any issues if uninstrumented code calls back into instrumented code?

3. On the statement level, is there some way that shadows could be updated incorrectly in the mixed-code execution?

We will consider these factors in the remainder of Section 3.2.2, and in summary, we will find that there are two concerns regarding accuracy. One issue is that calls back from uninstrumented code into instrumented code can unintentionally serialize these instrumented code calls at times when they should be treated as parallel calls, as is described in **Calls From Uninstrumented Code into Instrumented Code**, below. The other concern is that high coupling between user code classes can result in inaccurate behavior under certain circumstances, which is elaborated on in **Uninstrumented Code and Shadow Variables**, below.

In considering these factors, we will describe the instrumented code's reference to the dynamic analysis through a `DynamicAnalysis` object. This object simply provides the hooks that the instrumented code needs at the time of a dynamic analysis. One important feature that the DynamicAnalysis provides is the `DynamicAnalysis.LastTimeStamp`, which corresponds to the `last_calculated_timestamp` described in Section 2.4.3.

Calls from Instrumented Code to Uninstrumented Code

With uninstrumented libraries, it is important to consider the behavior of the analysis in the case that instrumented code calls into uninstrumented code. Let us examine the general form of an instrumented method, at the level of method instrumentation, to get an idea of the scope of its effect on the relationship of instrumented and uninstrumented methods. Recall that the general instrumentation for a method is presented in Figure 2.5 (an uninstrumented method) and in Figure 2.6 (its instrumented version). Additionally, it is relevant to review the instrumentation for a method call, which is defined in Section 2.4.3 and revisited for the purpose of this section in Figure 3.2.

```
// Uninstrumented, generic method call
someVal = m(a, b);

// Instrumented, generic method call
control_caller = DynamicAnalysis.ComputeStatementTime(control_caller, ws_a, ws_b);
//Note that DynamicAnalysis.lastTimeStamp is set by the above statement
someVal = m(a, b);
ws_someVal = DynamicAnalysis.GetLastTimeStamp();
//Gets the lasTimeStamp value set by the return of the callee
```

Figure 3.2: Instrumenting a call to method m().

From this overview of the instrumentation, it can be seen that there are few ties between method calls. The chief interaction of methods on this level is through `DynamicAnalysis.lastTimeStamp`. It is this variable that determines the callee's `control` value when it is invoked, and through this variable that the write shadow for the local on the left-hand side of a statement invocation (if any) is determined. In calling from an instrumented method into an uninstrumented method, the uninstrumented method does not receive the `lastTimeStamp` value, but it does not have a `control` variable to track, so this causes no unexpected loss in correctness; rather, this is the expected effect of analyzing an instrumented program with uninstrumented libraries. However, when the uninstrumented method call returns, it does not update the write shadow of the left-hand side (if any) of the invocation statement; this write shadow is instead set to the previously calculated *ComputeStatementTime(control, method parameters ...)*, thus creating a "unit cost" of one unit of time for the uninstrumented method call. This loss in accuracy is also not unexpected and is thus acceptable.

Another tie between method calls is exceptional control flow, but the relevant instrumentation does not impede correctness when interleaving instrumented and uninstrumented method calls. All of the high-level, original try-catch semantics of an instrumented method is preserved: exceptions that were caught and handled in a method are still dispositioned in the same manner, and exceptions that were thrown up from a method still do this. As such, any exceptions thrown from uninstrumented code back into instrumented code result in the same high-level control flow that they would have if all of the code were instrumented.

Calls From Uninstrumented Code into Instrumented Code

It is conceivable that library code can call back into user code, which means that we need to consider the behavior of uninstrumented code calling instrumented code. Fortunately, for the purposes of our measurements, we can assume that the dynamic analysis begins with an instrumented `WrapperMain` method and/or an instrumented benchmark top-level method, since this is the design of the experimental setup. This means that no matter where a program is in its execution, if there is a call from uninstrumented to instrumented code, then, at some earlier point, instrumented code has been executed.

When an instrumented method (IM) is executed, it will initialize its `control` value equal to the `DynamicAnalysis.lastTimeStamp`. For the first instrumented method called after a sequence of uninstrumented method (UM) calls, the `lastTimeStamp` would be relatively out of date (that is, smaller), as compared to its value at that point if all of the code had been instrumented. This `lastTimeStamp` was set at some earlier point in execution, just before the transition from instrumented code

to uninstrumented code, and although this value is out of date, this behavior is a reasonable and expected impact on correctness.

However, there is some issue with the processing of the `lastTimeStamp` if an uninstrumented method calls a sequence of multiple instrumented methods, as in the following example: (1) InstrumentedMethod1 calls UninstrumentedMethod1, (2) UninstrumentedMethod1 calls InstrumentedMethod2, then (3) UninstrumentedMethod1 calls InstrumentatedMethod3, ..., then (10) UninstrumentedMethod1 calls InstrumentatedMethod10. The behavior of this code, *assuming all code is instrumented and all methods return void*, is given in Figure 3.3. In this case, each time UM1 sets `lastTimeStamp`, its new value is one greater than its previous value.

```
IM1 sets DynamicAnalysis.lastTimeStamp
IM1 calls UM1
UM1 sets its control value to DynamicAnalysis.lastTimeStamp
// start at the value n

UM1 sets DynamicAnalysis.lastTimeStamp // to the value n + 1
UM1 calls IM2
IM2 sets its control value to DynamicAnalysis.lastTimeStamp
IM2 returns and sets DynamicAnalysis.lastTimeStamp to its updated control value

UM1 sets DynamicAnalysis.lastTimeStamp // to the value n + 2
UM1 calls IM3
IM3 sets its control value to DynamicAnalysis.lastTimeStamp
IM3 returns and sets DynamicAnalysis.lastTimeStamp to its updated control value
UM1 sets DynamicAnalysis.lastTimeStamp // to the value n + 3
UM1 calls IM4
IM4 sets its control value to DynamicAnalysis.lastTimeStamp
IM4 returns and sets DynamicAnalysis.lastTimeStamp to its updated control value
...

UM1 sets DynamicAnalysis.lastTimeStamp // to the value n + 9
UM1 calls IM10
IM10 sets its control value to DynamicAnalysis.lastTimeStamp
IM10 returns and sets DynamicAnalysis.lastTimeStamp to its updated control value
```

Figure 3.3: Behavior with all code instrumented.

```
IM1 sets DynamicAnalysis.lastTimeStamp // start at the value n
IM1 calls UM1

UM1 calls IM2
IM2 sets its control value to DynamicAnalysis.lastTimeStamp
IM2 returns and sets DynamicAnalysis.lastTimeStamp to its updated control value

UM1 calls IM3
IM3 sets its control value to DynamicAnalysis.lastTimeStamp
// n + some amount determined in IM2
IM3 returns and sets DynamicAnalysis.lastTimeStamp to its updated control value

UM1 calls IM4
IM4 sets its control value to DynamicAnalysis.lastTimeStamp
// n + some amount determined in IM2 and IM3
IM4 returns and sets DynamicAnalysis.lastTimeStamp to its updated control value
...

UM1 calls IM10
IM10 sets its control value to DynamicAnalysis.lastTimeStamp
// n + the given pattern
IM10 returns and sets DynamicAnalysis.lastTimeStamp to its updated control value
```

Figure 3.4: Behavior with UM1 method uninstrumented.

However, in the case that the uninstrumented code is actually uninstrumented, we have the behavior given in Figure 3.4. Here we see that a sequence of calls from uninstrumented code into instrumented code treats all individual calls back into instrumented code in a sequential manner, because an instrumented method expects the method that calls it to "reset" DynamicAnalysis.lastTimeStamp both after a method call and before the subsequent method call. To reiterate this in another fashion, the first time instrumented code is called from uninstrumented code, its behavior relative to DynamicAnalysis.lastTimeStamp is as would be expected in an execution in which all code is instrumented. However, once the instrumented code returns back into the uninstrumented code, a further call from the this uninstrumented code

into instrumented code will be treated as occurring sequentially after the first call into instrumented code, due to the fact that the `DynamicAnalysis.lastTimeStamp` value inherited by the second call into instrumented code is the same value set when the first call into instrumented code returns. This behavior is not immediately obvious and its potential serialization of instrumented code should be understood in performing analyses with uninstrumented code.

Uninstrumented Code and Shadow Variables

It would seem that there are few difficulties involved in how individual statement types are instrumented, in relation to executing uninstrumented code. (This is aside from how return statements are executed, which is taken into account in the preceding discussion.)

As described in Section 2.4.3, the instrumentation around individual statement types can either update the local `control` value, and/or update shadow variables, and/or update `DynamicAnalysis.lastTimeStamp`. In regards to correctness when using uninstrumented libraries, the update of a local `control` value is not an issue, as this value does not directly cross method call boundaries; the value contained therein is referenced by other method invocations only through the use of `DynamicAnalysis.lastTimeStamp`, which has been discussed previously.

At the first glance, updating shadow variables for local variables, static fields, instance fields, or array elements in a program with both instrumented and uninstrumented code does not seem to be a major issue, although this requires some thought. In instrumented code, the shadows are updated as expected, although they may not

be updated to an appropriately recent (that is, sufficiently large) timestamp if uninstrumented code has called into instrumented code, as mentioned in the preceding discussion.

However, it is possible to imagine that uninstrumented code can write to memory locations that have shadows, without updating those shadows. For example, suppose class `A_Inst` is instrumented, and class `B_Uninst` is not instrumented, in Figure 3.5. As in this example, it appears to be possible to have an uninstrumented class write to an instrumented class's field without updating any relevant shadows. Fortunately, it would seem to not be the case that any library classes would have knowledge of user code classes and be bound to them in this manner. Therefore, this possibility should not be an issue when leaving the libraries uninstrumented. It is more realistic, however, that an uninstrumented user code class may interact with an instrumented user code class in this manner, and as such there would be a risk of affecting the analysis' accuracy if the set of user code classes were to be only partially instrumented. Although "good design principles" would typically outlaw such coupling among classes, it is possible that the correctness of the analysis could be slightly influenced by such cases.

```
class A_inst {                     import mypackage.A_inst;
    public static int x;           class B_uninst {
    public static int rs_x, ws_x;  public void SomewhatContrived()
}                                  {
                                       A_inst.x = 5;
                                   } }
```

Figure 3.5: Two separate class files, one instrumented, one not.

3.2.3   A Note About Native Methods

As has been mentioned above, the entirety of the Java code in the libraries was instrumented for the analysis in Tables 3.3 and 3.4. However, it was not possible to instrument native methods that are called by the libraries using the same mechanism that instrumented the Java libraries and user code. Therefore, we have decided to treat any native methods calls as not influencing the dependences of the rest of the program, and as executing in unit time. The consequences of this decision are essentially the same as invoking uninstrumented Java methods, as has been discussed in this section.

3.3   Characterizing the Effects of Categories of Dependences

Above, we have explored the effects of *all* dependences — through static fields, instance fields, array elements, and local variables. It may be of further interest to characterize the effects of dependences through these individual categories of memory locations, which could help in understanding who much responsibility these broad categories bear in impeding potential parallelism. Here, we measure each benchmark with a modified version of the analysis, in which none of the dependences through either static fields, or through instance fields, or through array elements is considered. This will give us set of "ceiling values" for the potential parallelism available for that specific implementation of the benchmark, for that particular input. The highly optimistic speedup values returned will give us an idea of what sort of speedup might be achieved if a means were found to overcome the dependences in the given executions.

Table 3.6: Speedup: ignoring static/instance/array dependences.

Benchmark	Original Speedup	Speedup With Dependences Ignored Through:		
		Static Fields	Instance Fields	Array Elements
compress	1.2178	1.2179	3.3394	1.2326
db	1.1634	1.1634	25.6631	1.2148
jack	1.6865	1.7585	2.1232	1.7585
javac	2.7790	3.1422	20.9654	2.7340
jess	5.1821	5.1822	56.7372	5.1821
mpegaudio	1.6375	1.6375	2.1105	1.8718
Spec raytrace	2.4110	2.4110	121.7878	2.4110
euler	1.6424	1.6424	3.9534	1.8349
moldyn	1.8223	118.0800	1.8354	1.8223
montecarlo	1.2712	1.2712	298.4316	2.7018
JG raytracer	1.2211	1.2211	1.2692	1.2211
search	1.5775	1.5775	1.5775	1.5775
bh	47.4109	47.4130	47.6490	47.5226
bisort	1.1340	1.1341	1.1341	1.1340
em3d	3.9550	3.9550	5133.9577	3.9582
health	116.9337	116.9339	125.8781	116.9339
power	134.1112	134.1069	14168.5087	173.5981
tsp	38.1510	38.1509	40.8376	38.1509
voronoi	32.0687	32.0709	33.3974	32.0709
antlr	1.6256	1.6256	3.7026	1.6754
bloat	5.1387	5.1073	18.3595	6.0848
chart	2.0019	2.0173	5.8526	2.0053
fop	2.3843	2.3839	4.5079	2.3845
jython	2.2108	2.2144	2.5286	2.2118
luindex	3.9669	3.9665	7.2969	4.0359
pmd	2.0089	2.0111	4.0938	2.0161

3.3.1   Interpretations

As may be expected, static fields do not appear to be a bottleneck to parallelism in almost all of the benchmarks. This is reasonable, as non-final static fields can typically be expected to represent a small minority of the total set of memory locations in a Java program.

However, as can be seen from the results in Table 3.6, the interesting results begin to occur when dependences through instance fields or array elements are no longer considered. In the programs analyzed, the instance fields represent the most consistent and significant obstruction to potential parallelism. Array elements also presented a bottleneck, but not of the magnitude seen in instance fields. We can use this information as a starting point for more in-depth experiments, as will be seen in Chapter 4.

3.4   Characterizing the Effects of Essential Dependences

In Chapter 2, we discussed the three classic types of dependences: flow dependences, anti-dependences, and output dependences. Of these three types of dependences, only flow dependences are considered to be *essential*; that is, these are the only dependences inherent to the conceptual algorithm. In contrast, anti- and output dependences are consequences of an implementation. Non-essential dependences can conceivably by circumvented in a number of ways, including before execution time by a new implementation of a program, as well as at execution time by hardware. As such, it is of interest to characterize the effect of only essential (flow) dependences on potential parallelism, to better describe the extent in which non-essential dependences affect that potential parallelism.

Here, let us introduce a modified version of the analysis that only respects flow dependences, and by comparing it to the experiments from Section 3.1.2 that respect all dependences, we can describe the contribution of both essential and non-essential dependences to potential parallelism.

Table 3.7: Speedup: considering only flow dependences.

Benchmark	Original Speedup	Speedup With Flow Dependences Only	Speedup Increase (Absolute)	Speedup Increase (Percent)
compress	1.2178	1.2179	0.0001	0.01%
db	1.1634	1.1634	0.0000	0.00%
jack	1.6865	1.7717	0.0851	5.05%
javac	2.7790	2.8672	0.0882	3.17%
jess	5.1821	5.1968	0.0146	0.28%
mpegaudio	1.6375	1.6499	0.0125	0.76%
raytrace	2.4110	2.4112	0.0002	0.01%
euler	1.6424	1.6424	0.0000	0.00%
moldyn	1.8223	1.8224	0.0000	0.00%
montecarlo	1.2712	1.2712	0.0000	0.00%
raytracer	1.2211	1.2211	0.0000	0.00%
search	1.5775	1.5775	0.0000	0.00%
bh	47.4109	47.4255	0.0146	0.03%
bisort	1.1340	1.1340	0.0000	0.00%
em3d	3.9550	3.9550	0.0000	0.00%
health	116.9337	116.9566	0.0230	0.02%
power	134.1112	134.3901	0.2789	0.21%
tsp	38.1510	38.1520	0.0011	0.00%
voronoi	32.0687	32.0915	0.0228	0.07%
antlr	1.6256	1.7000	0.0745	4.58%
bloat	5.1387	6.3934	1.2547	24.42%
chart	2.0019	3.0245	1.0226	51.08%
fop	2.3843	2.6565	0.2722	11.42%
jython	2.2108	2.3144	0.1037	4.69%
luindex	3.9669	5.7224	1.7555	44.25%
pmd	2.0089	2.9131	0.9042	45.01%

3.4.1 Interpretations

These measurements indicate that non-essential dependences are not severely limiting the potential parallelism for many of the benchmarks. 23 of the 26 benchmarks showed a speedup increase of less than 1.0 when ignoring these dependences, and only three of the benchmarks showed a speedup increase of greater than 1.0. This suggests that the nature of the implementations is such that updates or upgrades to the algorithms that target anti- and output dependences (e.g., scalar expansion and array expansion) could not be expected to significantly improve potential parallelism.

However, as can be seen by the percentage increase in speedup, while the overall increase is often rather small, the relative increase as compared to the speedup respecting all types of dependences can be high. This is most consistently noticeable for the DaCapo benchmarks, which also happen to be the largest programs that were studied in this thesis. Future investigations are needed to provide more insights into these results.

Chapter 4: Case Studies

In this chapter, we will investigate several programs from the benchmark set presented in Chapter 3 and attempt to find components of their implementations that inhibit their potential parallelism. Using feedback from the analysis, assuming that all dependences as described in the coarse-grain parallelism model are respected, we will attempt to modify a benchmark through a series of changes to the source code, and in doing so improve its speedup, but preserve its sequential semantics. The benchmarks studied here have parallel versions, but the implementation details of the parallel versions are ignored (and in fact, this author did not review the parallel versions before performing each study) so that the analysis can be used to explore the benchmark without "knowing any answers in advance."

4.1   Java Grande `moldyn`

The first benchmark we will study is Java Grande's sequential `moldyn` program. As a small body of source code, it is very suitable to be studied first, to help demonstrate the manner in which the analysis can aid in finding impediments to parallelism.

4.1.1   Overview of the Benchmark

The benchmark (**Mol**ecular **Dyn**amics) simulates the interaction of particles in a three-dimensional space over a series of iterations. Its computation is iteration-based: in general, for each iteration, for each particle in the space, it determines the amount

of movement for that particle, and also determines the amount of force exerted on it by all of the other particles. It is a fairly small program, with a total of 5 classes and 23 methods (including the artificial `WrapperMain` added as part of the analysis's test harness). This limited size means that the effects of impediments to parallelism will be easier to localize than in a larger body of source code.

4.1.2   Summary of the Case Study

The benchmark shows a modest speedup of 1.8223. However, it is conceivable that we could reach a greater speedup if the implementation were improved. A comparison of this speedup, which respects all dependences, to the speedup for the benchmark as given in Table 3.3 where dependences through static fields are ignored, shows that the program has a significantly higher speedup of 118.0800 in that case. We will find that the key inhibition to speedup is a set of three static fields, and their usage in one loop is serializing a large part of the computation.

4.1.3   Experiments

After noticing the large difference between the speedup with all dependences considered and the speedup when ignoring dependences through static fields, our first step is to investigate the `moldyn` source code and find the static fields in its classes. This reduces our focus to 12 static fields, all members of the `md` class. Seven of these static fields are final, and thus cannot be written, so we discard the possibility of these fields impacting the speedup.

Of the remaining five non-final static fields, one is an array, three are `double`s, and one is an integer. It seems unlikely that the array is an impediment, since any dependences through its elements would not have influenced the speedup when

ignoring dependences through static fields, and intuition suggests that an array is a more parallelism-friendly datum than the remaining four scalar fields.

Code Transformation

Exploration: As described above, we have narrowed down the likely impediments to speedup to four static fields — the `double`s `epot`, `vir`, and `count`, and the integer `interactions`; all are initialized to zero. We will start by examining their use in the code.

Searching the program text for the variable name `interactions`, we find that it appears in only one other location: a method `force` in the class `particle`. This method is called a large number of times, representative of the entire three-dimensional problem space, and the method's body is a loop that has a number of iterations also representative of the entire the three-dimensional space. In addition to performing other calculations, it increments this `md.interactions` variable every loop iteration, provided some logical test passes. Similarly, the `md.epot` and `md.vir` static fields are updated in this loop, under the same conditions as `md.interactions`. A summary of this original code is presented in Figure 4.1.

Action: First, we created a new class, a container for an integer and two `double`s — that is, the static fields that are being updated in this `force` method. We then created an array with a number of these containers equal to the number of times that the `force` method was called, and passed an element of this array as a parameter into the `force` method. Then, instead of incrementing the static fields in the `force` method, it performed the same calculations on local variables, and just before the method returned, it assigned the values of these locals to the fields of the container instance.

70

```
// The benchmark's original top-level execution method, in the md class
public void runiters() {
    for (int a = 0; a < num_iterations; a++) {
        /* Do some work */

        for (b = 0; b < sizeOf3DSpace; b++) {
            particles[b].force( /*some parameters*/ );
        }

        /* Do more work */
    }
}




// In every call to the original force method, in the particle class,
// md's static fields are updated
void force(/*some parameters*/) {
    for (int i = 0; i < sizeOf3DSpace; i++)
    {
        /* Some calculations */
        if (some_test) {
            /* Some calculations */
            md.epot += /*some calculated value, not dependent on md.epot*/;
            md.vir -= /*some calculated value, not dependent on md.vir*/;
            md.interactions++;
        }
    }

}
```

Figure 4.1: Benchmark `moldyn`, original version.

Then, in the scope of the method that called `force`, after the loop that called `force`, a new loop was added in which the respective fields from the array of containers were summed. These sums were added back into the respective static fields `epot`, `vir`, and `interactions`, and this modified source code is summarized in Figure 4.2.

Intent: The repeated increment operation on `interactions`, as well as the repeated addition and subtraction performed on `epot` and `vir`, would seem to serialize the computations performed by the loop in the `force` method. Consequently, changing the operations on these static fields into operations on local fields, through which the dependences can be ignored, the computation could show greater speedup.

Result: The speedup improved significantly from the original value of 1.8223 to 102.7360, which is rather close to `moldyn`'s speedup of 118.0800 when completely ignoring dependences through static fields. This indicates that the changes have effectively removed the unwanted serialization through static fields.

4.1.4   Analysis of the Change

As has been established in the preceding section, the `moldyn` benchmark can be modified such that it shows a much higher speedup under the analysis described in this paper. In this section, let us briefly compare how these modifications correspond to changes that would be practical if a software designer would actually want to convert the original, sequential program into a parallel one.

The essence of the change is the following: instead of all of an iteration's calls to the `force` method accessing the same memory locations, each call to the `force` method is given its own small section of memory to store the persistent results of its calculations. At the end of an iteration, the work of each of the `force` method calls is

```
// The benchmark's modified top-level execution method, in the md class
public void runiters() {

    MD_Container[] containers = new MD_Container[sizeOf3DSpace];
    for (int a = 0; a < num_iterations; a++) {
        /* Do some work */

        for (b = 0; b < sizeOf3DSpace; b++) {
            particles[b].force( /*some parameters*/, containers[b]);
        }

        // Now re-calculate iterations, epot, and vir
        for (i=0;i<sizeOf3DSpace;i++) {
            interactions += containers[i].interactions;
            epot += containers[i].epot;
            vir += containers[i].vir;
        }

        /* Do more work */
    }
}

// The revised force method in the particle class,
// taking a container as a parameter
void force(/*original parameters*/, MD_Container mdc) {

    for (int i = 0; i < sizeOf3DSpace; i++){
        /* Some calculations */
        if (some_test) {
            /* Some calculations */
            mdc.epot += /*some calculated value, not dependent on md.epot*/;
            mdc.vir -= /*some calculated value, not dependent on md.vir*/;
            mdc.interactions++;
        }
    }
}

// A small container to hold the data computed by a call to force()
class MD_Container {
    int interactions;
    double epot, vir;
    MDContainer() {
        interactions = 0; epot = vir = 0.0;
    }
}
```

Figure 4.2: Benchmark `moldyn`, modified version.

reassembled into one set of data — the original implementation's three notable static fields — and then the next iteration of the program is ready to repeat these steps. This transformation is very similar to the standard *scalar expansion* optimization which is used to eliminate loop-carried dependences that inhibit parallelism, by introducing extra storage.

This new implementation uses more memory than the last, since it must allocate space to store the results of the computations in the `force` method calls during an iteration of processing. Exchanging a greater usage of memory in return for improved performance is not an unusual tradeoff in making parallel programs, however, so this could likely be an acceptable change to be made in preparation to make a parallel version of the program.

Let us make one note about the semantics of this program and the consequences of our change. If one is to assume that floating-point addition is associative, in which case $(a + b) + c$ is identically equal to $a + (b + c)$, then this change preserves the semantics of the original program. However, it is the case that an actual implementation of floating-point addition could potentially yield very minor differences when re-associating addition computations in this way. Conceptually, there is no strong reason that the original ordering of the addition computations that yields the final values for `epot` and `vir` is "more correct" than our reordering, but we recognize the possibility that some specific users of software may be more concerned with exactly preserving pre-established floating-point results and that they would not be comfortable with even the slightest change in such a result.

4.2 SPEC JVM98 `raytrace`

The SPEC JVM98 raytracer benchmark receives an unassuming speedup score
of 2.4110 in the initial analysis. This is somewhat less than could be expected, in
that the author's understanding of a basic, one-pass raytracer algorithm is that it
has a very high potential parallelism: the features of a basic raytracer can involve
many reads from objects that compose a scene, and writes only to individual pixels
on a canvas that are not dependent on one another. Let us explore this program to
see why this implementation does not show appreciable potential parallelism on its
100-sized input.

4.2.1 Overview of the Benchmark

A raytracer is a computer graphics tool that renders a two-dimensional image
(a *canvas*) from a set of three-dimensional models (a *scene*). Typically, a simple
raytracer such as the one included in the SPEC JVM98 benchmark suite will contain
the capability to render simple geometrical objects, such as spheres or prisms, given
points of light that illuminate the objects. An *eye* or *camera* represents the point of
view that is observing the scene, including the position of the eye, the direction it is
looking, its orientation (that is, what direction is up) and the width and height of its
field of view. At its most basic level, the general algorithm involves the casting of *rays*
from the eye through each pixel on the canvas; the point at which the ray collides
with an object determines the color of that pixel. Finding whether this point on the
object is in shadow, relative to a given light, is determined by casting a ray from this
collision point to that point light source and determining whether any other scene
objects exist on the path of that ray in between this object and the light. Reflective

color can be calculated by casting a ray from the object from the point of the eye ray's collision and finding the color of the object of what the new ray collides with; this reflective recursion is limited to some depth so that the reflective color calculation eventually terminates. Finally, the components of the color (the object's color times the intensity of the intensity of the lights that are visible to it, plus reflective color, and color from any more advanced features) are summed to generate the color for the given pixel in the canvas.

As described above, this raytracer should be expected to benefit heavily from parallel processing. The algorithm writes color values to individual pixels in the final image (a two-dimensional array) and the color of any one pixel does not affect the color of any other pixel; the calculation of a pixel color is thus dependent upon many reads of scene object colors and light positions, but not on values that are written during the computation. The value of examining this benchmark is in determining whether the analysis described in this paper can unearth this potential parallelism, or in whether it can find an implementation bottleneck and help to explain the existence of that bottleneck.

A parallel version of a raytracer can be made manually by simply assigning separate program threads to separate rectangular divisions of the output canvas. Since the potential parallelism should in theory be proportional to the number of pixels in the canvas, it is expected that we can find more potential parallelism in this benchmark.

4.2.2 Summary of the Case Study

The sequential raytracer did not reveal much potential parallelism as it was initially implemented, but a design change did yield a noticeable increase in speedup. In

the original implementation, the raytracer rendered the entire scene (all of the pixels) as part of one large method call; here, we have changed the benchmark to render its scene by dividing the workload among an arbitrary number of method calls, so that each draws a portion of the canvas. Given our model of coarse-grain parallelism in which individual method calls can be run concurrently, this makes it possible to view more of the potential parallelism in the algorithm.

4.2.3   Experiments

Here, we alter the benchmark in an effort to unearth an implementation with greater potential parallelism, and our creation of a new version with a higher speedup indicates success in proportion to the increase in speedup. Recall that it is necessary for any changes to have the same semantics as the original version of the program; otherwise, finding a change in speedup has little meaning.

Code Transformation

Exploration: The design of the program involves calling one method, `Scene.RenderScene`, to perform the main body of work in the benchmark. `RenderScene` takes four parameters: the canvas on which to draw, the width of the canvas, the number of sections into which the work has been divided, and the section for which this call to `RenderScene` is responsible.

Action: We added multiple calls to the rendering method, the number of which can be specified by a command-line argument. Each call to `RenderScene` was passed a unique subsection of the canvas to process, and there were a number of calls equal to the number of subsections.

Intent: By breaking up the initially large amount of work into several smaller portions of the problem, the theoretically parallel version of this program may be able to run each of these rendering calls in parallel, with no dependences between them.

Result: The speedup increased noticeably as the work was divided into multiple intervals of the canvas. Table 4.1 shows the progression of speedup along with the number of calls made to the `RenderScene` method during a program execution.

Separations of Canvas	Speedup	
	SPEC JVM98 `raytrace`	**Java Grande** `raytracer`
1	3.1382	1.2940
2	4.6534	2.4500
3	6.9526	2.1510
4	6.4241	3.0750
5	7.1539	3.4481
6	8.3545	4.0527
7	7.6358	5.1606
8	8.7444	5.2048
9	8.5300	6.1885
10	8.6822	6.4300
11	9.1799	7.5679
12	8.4629	7.5758
13	8.6099	8.7847
14	8.3078	8.8210
15	7.9334	8.8286
16	7.5741	11.1006

Table 4.1: Speedup from dividing the workload into separate render calls for the raytracers.

4.2.4 Analysis of the Change

In this experiment we have attempted to simulate the separating of the canvas drawing into separate threads by separating work among method calls. A basic

raytracer represents an algorithm that can theoretically be made parallel at the level of calculating colors of individual pixels, so we expected that dividing the work into these separate method calls would reveal more potential parallelism. We have found that the implementation of the SPEC JVM98 `raytrace` is such that potential parallelism can be revealed by moving away from an implementation in which the entire canvas is drawn by one method call, to an implementation in which smaller portions of the canvas a drawn by separate, non-dependent method calls. These separate method calls could easily translate into separate threads in an actual parallel version of the program. Although this move is approaching the per-pixel potential processing that a theoretical algorithm could exhibit, note that in Table 4.1, we find that there are appears to be a peak speedup value at about 11 divisions of the canvas. This means that in our concrete `raytrace` implementation, the potential parallelism does not seem to reach a maximum when the work is divided into extremely small portions, as would be projected from the conceptual algorithm.

There are two reasons that could cause this behavior, both related to overhead. The first is that once the canvas has been divided up into enough sections, there is not enough work available for a given `RenderScene` call to process, in proportion to the amount of code that needs to be executed to make a call to that method. There are a number of statements, including local variable initialization, canvas section dividing, and loop control variable incrementing and testing that will be executed during any call to `RenderScene()`, and with less and less real work for the method to perform, the greater the impact of this overhead code on speedup. The second reason could be implementation details that create artificial bottlenecks, such as a library call or helper method storing data in a persistent memory location to which access must be

serialized among calls to it. While there are no immediately obvious methods that offend in this way in this raytracer, we will see that there are some such issues in the Java Grande `raytracer` in Section 4.3.

4.3   Java Grande `raytracer`

The Java Grande raytracer benchmark receives a low speedup score of 1.2211 in the initial analysis, and the further experiments performed on it that ignore different categories of dependences do not noticeably raise this score. Given the above discussion of the nature of a raytracer algorithm, these results are counterintuitive and deserve further analysis.

4.3.1   Overview of the Benchmark

Much like the SPEC JVM98 `raytrace` analyzed above in Section 4.2, the Java Grande Sequential `raytracer` is a basic raytracer that supports simple lighting, shadows, and reflection. Its implementation is slightly different than the SPEC one, and instead of actually reading in input from a file, the Java Grande program uses a hard-coded input scene.

4.3.2   Summary of the Case Study

Early tests did not reveal any potential parallelism beyond the speedup value found in the initial analysis that respected all dependences. Starting from the same point as in the SPEC JVM98 `raytrace` study, we attempted to alter the benchmark code to make multiple calls to the rendering function, thus dividing the workload by passing each call different pixel spaces on the canvas. However, this did not increase the potential parallelism discovered by the analysis, so this implementation of a raytracer

appeared to have additional impediments to parallelism. It was then found that the main impediments to parallelism were a serializing checksum computation and the use of class instance variables as "persistent temporary variables" that the code comments said were intended to help "speed up" sequential computation by removing the need to declare temporary variables in class methods.

4.3.3 Experiments

In the following experiments, we incrementally alter portions of the program under test and run the analysis (respecting all dependences) after each change is made, to find the changes these alterations make to the program's speedup score. A higher speedup indicates that the alteration has increased the program's potential parallelism. Recall that it is necessary for these changes to have the same semantics as the original version of the program; otherwise, finding a change in speedup is meaningless. Unlike the `moldyn` benchmark, the speedups when ignoring categories of dependences do not provide us with specific insights, so we will begin by analyzing the high-level structure of the target program.

Change 1

Exploration: The design of the program involves calling one method, `Raytracer.render`, to perform the main body of work in the benchmark. Method `render` takes one parameter: an interval of the image to construct.

Action: We added multiple calls to the rendering method, which performs the real work of the raytracer. The calls took a parameter to describe the range of the canvas over which they would render, so each call received a unique subsection of the canvas. The change is shown in Figure 4.3.

```
//Original: Render the entire canvas in one method call
Interval interval = new Interval(/*entire problem space*/);
render(interval);

//New: Render the canvas in N method calls
Interval[] intervals = new Interval[N];
for (int i = 0; i < N; i++) {
    intervals[i] = new Interval(/*vertical section of problem space*/);
    render(intervals[i]);
}
```

Figure 4.3: Breaking up the Java Grande sequential `raytracer`'s workload.

Intent: By breaking up the initially large amount of work into several smaller portions of the problem, the theoretically parallel version of this program may be able to run each of these rendering calls in parallel, with no dependences between them.

Result: The speedup remained largely the same at 1.2210, even when the work was divided among 10 intervals. There must be dependences that are not visible to the high-level method that invokes the rendering method, and we will have to look deeper into the structure of the program. For the rest of the changes, we will perform the work over 10 separate intervals.

Change 2

Exploration: `Raytracer.render` iterates over every pixel in the interval, calculating its color. For each pixel, just after that pixel's color is calculated, it adds the numeric value of that color components (three eight-bit integers) to a `Raytracer.checksum` integer instance field.

Action: We moved the checksum computation from `Raytracer.render` to the method that calls `render`, just after all of the image has been processed.

82

Intent: The addition of color components, all written into a single memory location for every pixel, would seem to serialize the computation, and this serialization must be circumvented. Given the nature of the checksum, its calculation could simply be removed from the rendering method and placed at the end of the entire computation.

Result: The speedup remained mostly the same at 1.2211, even when the work was divided among 10 intervals. There must be further dependences at work.

Change 3

Exploration: The `Raytracer` class has 14 instance fields, but two stand out in the code comments: the `Raytracer.tRay` ray and `Raytracer.L` vector are described to be "temporary" variables. Inspecting the usage of `L`, it does in fact appear to be "temporary" in the sense that its value is set in the scope of a single method call, never used again after that call returns, and not altered by any other methods during its usage. Inspecting the usage of `tRay`, it does not actually appear to be a "temporary variable" in the classic sense, since its fields are read and written outside of the scope of a method in which they are initially defined.

Action: We removed the object-wide declaration of the `L` temporary variable, and placed its declaration into the `Raytracer` method `shade` that uses it. Pseudocode for this change is given in Figure 4.4.

Intent: In the original program, we have an object on the heap pointed to by `L`, which has three fields (`double`s `x`, `y`, and `z`). The lifetime of the values in these fields is only within the scope of a call to `shade`, and the memory locations named by these fields are repeatedly overwritten by subsequent calls to `shade`. This change creates

83

separate objects with separate `x`, `y`, and `z` memory locations that correspond to the lifetime of the values used by invocations of `shade`.

Under typical circumstances, it is not a justifiable action to define a class-level temporary object, since each instance of its usage is specific to some computation that occurs during one method call. Further, in a parallel program, it does not make sense for one memory location to be shared among multiple processes as a place for frequently altered temporary data.

Result: As before, the speedup remained mostly the same at 1.2210, so there must be further dependences at work. In the code comments, the "class-scoped temporary variables" are declared for "speedup;" this suggests that the original implementers viewed the construction and destruction of temporary objects to be expensive enough that removing those steps would improve the speed of the sequential computation.

Change 4

Exploration: The `Sphere` class has five instance fields, but just as in the `Raytracer`, two stand out in the code comments: the `Sphere.v` and `Sphere.b` vectors are described to be "temporary" variables, and in fact `Sphere.v` is used in that manner. `Sphere.b` is actually never accessed, although if it were, it would seem that it had been intended to follow the same usage pattern as `Sphere.v`.

Action: We removed the object-wide declaration of these two temporary variables and placed their declaration into the `Sphere.intersect` method that uses them. Pseudocode for this change is given in Figure 4.5.

Intent: This modification has the same intent as Change 3.

84

```
// Original Version

class Raytracer {
 Vec L = new Vec(); // Vec is a triple of 'double' values (x, y, z coordinates)
 // Raytracer's method to compute a color at a given point on a surface
 private void shade( /* some parameters */ ) {
    /* Some calculations */
    // Values of this.L.x, this.L.y, this.L.z reset with new values
    this.L.sub2(someVector1,someVector2);
    // dotProduct() reads from L's members
    if (dotProduct(this.L, someVector3) >= 0) {
        // normalize() writes to L's members, returns length
        double t = this.L.normalize();
        /* One read from L's members */
        // Nothing else accesses L anymore
        /* Some calculations */
    }
 }
}

// Modified Version

class Raytracer {
 // No field L
 // Raytracer's method to compute a color at a given point on a surface
 private void shade( /* some parameters */ ) {
    /* Some calculations */
    // Create a new Vec object specific to this invocation of shade
    Vec L = new Vec();
    L.sub2(someVector1,someVector2);
    // dotProduct() reads from L's members
    if (dotProduct(L, someVector3) >= 0) {
        // normalize() writes to L's members, returns length
        double t = L.normalize();
        /* One read from L's members */
        // Nothing else accesses L anymore
        /* Some calculations */
    }
 }
}
```

Figure 4.4: Java Grande **raytracer**, Change 3.

Result: The speedup makes a very modest increase to 1.2936 with the work divided among 10 intervals. This is the beginning of an improvement, but not on the order of magnitude that is expected, so there must be further dependences at work.

Unlike the negligible difference between Change 3's speedup and the original implementation, here we find an improvement in speedup that demonstrates that the practice of using an object-scoped temporary variable can measurably impede potential parallelism.

Change 5

Exploration: The `Raytracer` class has another instance field with a conspicuous code comment named `Raytracer.inter`. It is described as the "current intersection instance" and that "Only one is needed!" Inspecting how this field is used, it is not a temporary variable like the others analyzed to this point, as the lifetime of the object that it points to crosses method call boundaries.

Action: We removed the class-wide declaration of `Raytracer.inter` and declared it in `Raytracer.trace`, which is responsible for defining it in the original program. Method `Raytracer.intersect` uses the value of `inter` computed in `trace`, so `inter` was added as a parameter to this method. Pseudocode for the original version of the code is given in Figure 4.6, and the modifications are given in Figure 4.7.

Intent: The intersection object `inter` was being stored and accessed on the class level when it could just as easily have been declared and defined local to the method in which it was computed, and then passed as a parameter to callees that required it. Declaring and defining the `inter` object locally is more friendly to potential parallelism. This is because it allows for separate `intersect` calls to write to different `Isect` instances, and thus separate memory locations, whereas all of the `intersect`

86

```java
// Original Version

// Sphere Constructor
 public Sphere( /* some parameters */ ) {
    /* initialize other fields */
    this.v = new Vec(); //defined once
}

// Sphere intersection detection method
public Isect Intersect( /* some parameters */ ) {
    // Values of this.v.x, this.v.y, this.v.z reset with new values
    this.v.sub2(someVector1,someVector2);
    /* some reads from v */
}



// Modified Version

// Sphere Constructor
 public Sphere( /* some parameters */ ) {
    /* initialize other fields */
    /* No field v */
}

// Sphere intersection detection method
public Isect Intersect( /* some parameters */ ) {
    // Create a new Vec object specific to this invocation of Intersect
    Vec v = new Vec();
    // Assign freshly calculated values v.x, v.y, v.z
    v.sub2(someVector1,someVector2);
    /* some reads from v */
}
```

Figure 4.5: Java Grande **raytracer**, Change 4.

calls had previously written to the same memory locations, which artificially serialized them.

Result: The speedup makes a slight increase to 1.2984 with the work divided among 10 intervals. so there must be further dependences at work.

```
class Raytracer {

 // An Isect object contains a double, an int,
 // and pointers to a scene-object and that object's surface
 Isect inter = new Isect();

 // Raytracer intersection detection method
 public boolean intersect(Ray ray) {
     this.inter.t = /* a large, constant double value */
     for (int i = 0; i < numberObjectsInScene; i++) {
         if ( /* intersection is closest detected so far */ ) {
             /* write to several of this.inter's fields */
     }   }
 }

 // Raytracer method for firing a ray and finding what it hits.
 // Calls intersect()
 private Vec trace( /* some parameters */, Ray r ) {
     //Find what the ray r intersects, if anything
     boolean hit = intersect(r); //this.inter's fields are defined
     /* Perform some reads on inter's fields */
 }

 // Raytracer method for determining shadow.  Calls intersect()
 public int Shadow(Ray r) {
     if (intersect(r)) return 0; else return 1; }
}
```

Figure 4.6: Java Grande **raytracer**, Change 5, original version.

```
class Raytracer {

 // No field inter

 // Raytracer intersection detection method
 public boolean intersect(Ray ray, Isect inter) {
    inter.t = /* a large, constant double value */
    for (int i = 0; i < numberObjectsInScene; i++) {
        if ( /* intersection is closest detected so far */ ) {
            /* write to several of inter's fields */
    }   }
 }

 // Raytracer method for firing a ray and finding what it hits.
 // Calls intersect()
 private Vec trace( /* some parameters */, Ray r ) {
    Isect inter = new Isect();
    //Find what the ray r intersects, if anything
    boolean hit = intersect(r, inter);  //inter's fields are defined
    /* Perform some reads on inter */
 }

 // Raytracer method for determining shadow.  Calls intersect()
 public int Shadow(Ray r) {
    if (intersect(r, new Isect())) return 0; else return 1; }
}
```

Figure 4.7: Java Grande **raytracer**, Change 5, modified version.

Change 6

Exploration: Earlier, we briefly inspected the use of the `Raytracer.tRay` field, and saw that its use crossed method boundaries. Inspecting it further, it can be re-scoped as a local in the `render` method. The object is used in `trace`, which is called by `render`, as well as in `shade`, which is called by `trace`. There is some recursive use of the object between `trace` and `shade`.

Action: We changed the declaration of `Raytracer.tRay` from an instance field to a local variable of `render`, and added a corresponding parameter to `trace` and `shade`. Method `render` now passes the object to `trace`, and `trace` passes it to `render`, sometimes with further recursion from `render` back to `trace` and so on. Pseudocode for the original version of the code is given in Figure 4.8, and the modifications are given in Figure 4.9.

Intent: The use of `tRay` in the recursive sequence of `trace` and `shade` calls from `render` was inadvertently serializing the multiple calls to `render`. Essentially, there are a plethora of read operations and write operations on tRay's two fields as a result of a call by `render` to `trace`, and this means that subsequent calls to `render` cannot run in parallel. That is, in the second of two calls to `render`, the first operation that accesses a field of `tRay` must be scheduled near the time of the return of the previous call to `render`, when that `tRay` field was last written. By making the `Ray` class instance local to a call to `render`, which creates separate memory locations for each `render` call to access, there is no longer serialization on `tRay`'s fields between consecutive `render` calls.

Result: The speedup of the program jumped from the initial value of 1.2211 to 6.4300 for splitting the work over 10 intervals. This is an order-of-magnitude increase

is consistent with what the author would expect from the potential parallelism in a raytracer algorithm. Speedup values for dividing the canvas among one through sixteen intervals are presented in Table 4.1.

4.3.4 Analysis of the Changes

This benchmark was more difficult to modify in a way that revealed more potential parallelism because it had multiple contributing factors that inhibited that parallelism. Here, we began our analysis by reviewing the source code, and we altered what appeared to be the greatest serializing factors first: there was an opportunity to break up the work among `render` calls, and there was a checksum computation that took place after each pixel was calculated. After this low-hanging fruit was picked, we investigated the code comments to find the purpose of individual fields and their usage. This resulted in finding several instance fields that appeared to be used in a parallelism-unfriendly manner, and thus their replacement with the use of semantically equivalent local variables corresponded more closely to a parallel design. Finally, we reviewed the use of other instance fields and found that their usage among the raytracer's methods was local to a hierarchy of method calls — a method would define a value and pass it to its callees, never to use that value again after they returned. By converting these instance fields that point to long-living objects into locally scoped variables that pointed to shorter-living objects that were passed among method calls, we limited the wide scope of the memory locations that were being accessed and ultimately saw a large increase in speedup.

For each change, we used the analysis as a guide to verify its effects on potential parallelism, and in this way we converted the program into a version that was still

91

sequential, but better primed to take advantage of a multiprocessing environment if it were to be converted into a parallel program. Note that we have retained the original program's semantics through the entire sequence of changes, and that this was straightforward to do because of the checksum computation built-in to the benchmark. In a program without such a checksum, but with this many or more changes on the path to showing more potential parallelism, it would be necessary to find another way to easily verify that the semantics had not changed, such as automated batch tests on a variety of inputs.

```
class RayTracer {
 Ray tRay = new Ray();  //A ray contains two vectors: a point P and a direction D

 // Raytracer method that produces image for an interval of the problem space
 public void render( /* some parameters */ ) {
        Ray ray;
        for (y = interval.y_start; y < interval.y_end; y++) {
            for(x = 0; x < interval.width; x++) {
                ray = /* A ray from the camera in the direction of this pixel */
                trace( /* some parameters */, ray);
            }
        }
 }

 // Raytracer method for firing a ray and finding what it hits
 private Vec trace( /* some parameters */, Ray r)
 //When called by shade(), note that tRay and r point to the same object
    boolean hit = intersect(r); //Does the ray hit anything?
    if (hit) {
        /* Read from r's fields */
        shade( /* some parameters */, r.D );
    }
 }

 // Raytracer method for determining color at point in space
 private Vec shade( /* some parameters */, Vec I) {
 //Note that I and this.tRay.D currently point to the same object
    /* Read from I's fields */
    /* Perform some writes to the Vec objects this.tRay.P and this.tRay.D */
    if (some_test1) { /* read this.tRay and its fields */ }
    if (some_test2) {
        /* Read from I's fields */
        this.tRay.D = /* some vector */
        trace( /* some parameters */, this.tRay);
    }
    if (some_test3) {
        /* Read from I's fields */
        if (some_test4) this.tRay.D = /* some vector */
        else this.tRay.D = /* some other vector */
        trace( /* some parameters */, this.tRay);
    }
 }
}
```

Figure 4.8: Java Grande **raytracer**, Change 6, original version.

```
class RayTracer {
 //No field tRay

 // Raytracer method that produces image for an interval of the problem space
 public void render( /* some parameters */ ) {
       Ray ray;
       Ray localizedRay = new Ray(); // Add a local ray to replace tRay here
       for (y = interval.y_start; y < interval.y_end; y++) {
           for(x = 0; x < interval.width; x++) {
               ray = /* A ray from the camera in the direction of this pixel */
               trace( /* some parameters */, ray, localizedRay);
           }
       }
 }

 // Raytracer method for firing a ray and finding what it hits
 private Vec trace( /* some parameters */, Ray r, Ray tRay)
 //When called by shade(), note that tRay and r point to the same object
    boolean hit = intersect(r); //Does the ray hit anything?
    if (hit) {
        /* Read from r's fields */
        shade( /* some parameters */, r.D, tRay);
    }
 }

 // Raytracer method for determining color at point in space
 private Vec shade( /* some parameters */, Vec I, Ray tRay) {
 //Note that I and tRay.D currently point to the same object
    /* Read from I's fields */
    /* Perform some writes to the Vec objects tRay.P and tRay.D */
    if (some_test1) { /* read tRay and its fields */ }
    if (some_test2) {
        /* Read from I's fields */
        tRay.D = /* some vector */
        //Redundant parameters passed for the sake of clarity
        trace( /* some parameters */, tRay, tRay);
    }
    if (some_test3) {
        /* Read from I's fields */
        if (some_test4) tRay.D = /* some vector */
        else tRay.D = /* some other vector */
        //Redundant parameters passed for the sake of clarity
        trace( /* some parameters */, tRay, tRay);
    }
 }
}
```

Figure 4.9: Java Grande **raytracer**, Change 6, modified version.

Chapter 5: Related Work

As has been mentioned previously, the primary source of inspiration for this work was authored by Kumar in his analysis of potential parallelism in FORTRAN programs [4]. He introduced the notion of statements executing at a logical time, and used a very similar model for recording data dependences. Kumar's work differed in several ways from ours. The focus of his work are scientific and engineering applications written in Fortran. The model of parallelism is fine-grained (i.e., individual statements execute concurrently) and is suited for instruction-level parallelism. The approach is designed for analysis of arrays and scalars and does not handle dynamically allocated memory, pointer-based data structures, and user-defined types (e.g., structure types and class types). The experimental evaluation on four applications finds a significant degree of best-case parallelism: speedups of one to three orders of magnitude are observed, likely because these are scientific applications with loop-based array-based computations. In contrast, we consider coarser-grain parallelism at the method level (as appropriate for Java), and find that most of our benchmarks do not exhibit a significant degree of potential parallelism.

Austin and Sohi took the concept of exploring fine-grain parallelism further in [1]. They introduce a Dynamic Dependence Graph (DDG) that represents a single program execution by partially ordering its instructions based on data dependences. The DDG's height to total nodes ratio represents a notion similar to the concept of speedup that is used in this thesis. Although the analysis in this paper always respects output-

and anti-dependences, the work in [1] explores the potential parallelism in an execution by assuming that only flow dependences must always be respected, and that register renaming and memory renaming can sometimes be used to further improve parallelism. Since Austin and Sohi's work is concerned with fine-grain parallelism at the instruction level, it differs from this thesis in that it assumes that different types of instructions will take different amounts of time to execute (for example, a multiplication instruction will take more cycles than an addition instruction), whereas we are focused on a higher coarse-grain level of parallelism and thus assume that all Jimple statements take the same amount of time to execute.

Larus explored loop-based potential parallelism in [6]. The model of parallelism used in that paper differs from Kumar's and Austin and Sohi's fine-grain approaches in that it focuses exclusively on executing loop iterations in parallel. Like the previously mentioned approaches, and like this thesis, Larus's work executes a program once on a particular input, and describes the potential parallelism properties of that single execution. Unlike the previously mentioned approaches, but still like this thesis, Larus's model accepts a serialization of some part of the body of source code (all of the code outside of loops, whereas this thesis accepts the serialization of code within a method) and treats a certain part of that source code as a construct that can be improved via parallelization.

Rauchwerger also explores potential parallelism in [10] and discusses the question of how much practical parallelism can be extracted from it. His paper distinguishes dependences that cannot be circumvented (flow and control) as essential dependences, from all others as resource dependences. In addition to the manner used the previously mentioned papers and this thesis, which uses what he terms a "greedy" scheme in

which statements are scheduled to execute at the earliest logical time possible, his work also introduces a "lazy" scheme in which statements are scheduled to run at the latest time possible. Like Austin and Sohi, his work makes use of a dynamic dependence graph crafted over an execution. Like this thesis, the Rauchwerger paper assumes that statements execute in unit time; unlike this thesis, which assumes calls to native methods do not access data that has been shadowed, it treats system calls as potentially accessing any data computed by the time of the call, and thus artificially serializes the computation around them.

Postiff also makes an explicit distinction between essential and non-essential dependences in [9]. In the context of instruction-level parallelism (ILP), he finds that an impractically large, but finite, instruction window is not good enough to find parallel instructions, which tend to be further apart than even a very optimistic finite-sized window can hold. He suggests that a multiprocessor is needed to unearth the parallelism between these remote instructions, and this relates to this thesis' motivation to explore method-based coarse-grain parallelism in programs.

Also in the context of ILP, Wall presents a model [12] of architecture-level dependences such as registers, memory locations, and branches, and discusses the means by which a processor or compiler could expose parallelism in the presence of such dependences. He found that ambitious models of parallel hardware did not expose great instruction-level parallelism, and that having a large but finite number of processors (in the range of 256) did not significantly affect instruction-level parallelism unless other assumptions about ignoring anti- and output dependences and branch prediction were perfect. This last finding is relevant to our work in that this thesis makes assumptions like this in generating a speedup score.

Lam and Wilson further explore the affect of branch prediction on ILP [5]. They recognize that processors that make branch predictions typically throw away all of the instructions already executed on a predicted branch that is found to be not taken, even if some of these instructions would be reached and executed anyway due to other control flow. In presenting multiple models in which a theoretical machine is capable of some combination of speculating infinite consecutive branch outcomes, control dependence analysis, and following multiple contexts of execution, they found that control flow severely limits potential parallelism. This finding may be relevant to future investigations of more refined variations of our model of parallelism.

In [7], Mak and Mycroft attempted to unify the work performed by Wall, Austin and Sohi, Postiff, and Lam and Wilson. Their approach was to measure potential parallelism by generating a dynamic dependence graph based on a choice of what types of dependences to include as edges. These dependences types include flow dependences, name dependences (output and anti-dependences), control dependences, dependences based on address calculations, and flow dependences based on the stack pointer. They find that control dependences tend to restrict potential parallelism, with the caveat that branch prediction helps to circumvent this limitation within limited ranges of instructions, and that programs can be artificially serialized by the compiler through the use of the stack pointer. By introducing the concept of the spaghetti stack to avoid dependences based on frame allocations, and by distilling the dependences down to the "essential" flow dependences, they provide a measurement of the potential parallelism of the conceptual algorithm behind the program. This is somewhat related to the focus of our work, in which we are interested in implementation-based dependences that affect program design, but also the potential

to transform that design into one that more closely relates to the potential parallelism inherent in the conceptual algorithm.

Many of these works explore machine-code languages and discuss architecture-related parallelism concepts like register renaming or memory location renaming, or even branch prediction. These approach are intended to avoid anti- and output dependences in promoting instruction-level parallelism, or to avoid lost cycles due to unknown future control flow, as would be appropriate in exploring compile-time optimizations or similar enhancements that relate more closely to the hardware that could execute an algorithm. However, the analysis in this thesis focuses on providing more high-level information that can be directly useful to the programmer. We consider method-level parallelism instead of instruction-level or loop-level parallelism, and focus on handling all Java features such as dynamically allocated memory, pointer-based data structures, and user-defined types. Rather than focusing on anti/output dependences and branch prediction, we are concerned with finding the high-level program entities (e.g., fields and methods) that the programmer should examine in order to understand and improve the design-level parallelism properties of the program.

Chapter 6: Conclusions and Future Work

In this thesis, we have examined the potential parallelism in 26 sequential Java benchmarks by defining a coarse-grained model of parallelism and introducing a combination of a static analysis and a dynamic analysis to measure that parallelism. In respecting data dependences through high-level program design concepts such as static fields, instance fields, and array elements, as well as through sequential control flow within a method, we found that although some benchmarks showed optimistic two- or three-digit speedups, most of the benchmarks showed a low speedup in the range of one to six. This discovery meant that many of these programs have inherent properties of their implementations that prevent them from being candidates for an easy design change to parallel programs.

We also examined individual sources of data dependences, including static fields, instance fields, and array elements, and found that these categories of memory locations have different magnitude of impacts on potential parallelism. We found that static fields generally had a very minimal effect on potential parallelism, whereas array elements sometimes had a noticeable contribution to it, and instance fields could have significant effects on speedup. Depending on the program, ignoring dependences through instance fields could raise the speedup by one or more orders of magnitude, illustrating the large contribution of instance fields to impeding potential parallelism.

We then studied the effect of the Java libraries on our measurements. We found that measurements that consider only user code and do not measure the contributions of the libraries incur a significantly lower overhead than those experiments that measure both user code and library code. However, the speedup values gathered in these circumstances can vary significantly from those in the prior measurements, and at this time we are not aware of any consistent cause for a rise or drop in speedup when the libraries are not measured. We further examined the effect of essential dependences on potential parallelism, in the absence of non-essential dependences. In general, we found that in terms of absolute speedup, the potential parallelism in each of the benchmarks was not significantly affected by non-essential dependences, although in terms of relative speedup, some programs showed a fair increase in potential parallelism under these assumptions.

Finally, we studied several of the benchmarks in more depth and made changes to them, all while preserving the original semantics of those programs. Using the analysis as a guide, we were able to hand-alter these benchmarks into versions that showed higher speedup, and thus greater potential parallelism. In this manner, we have used the analysis as a form of feedback in performing design-level changes to existing sequential programs to prepare them to be altered in parallel versions.

This work can be expanded in the future by performing more case studies with further programs, which may reveal other ways in which programs can be altered to expose additional potential parallelism. Also, because it can be difficult to localize an inhibition to speedup in a large body of source code, it would be of interest to apply disciplined techniques that analyze a program's structure to automatically explore the program and recommend places that show the most promise in being altered to

improve potential parallelism. Another way to expand the work would be to change our assumptions about the model of parallelism, such as by assuming that only certain method invocations can be run in parallel, or that only certain numbers of methods may be run in parallel at any given time. Such changes may give further insight as to the potential parallelism of a program if only major portions were altered to include threading, or as to how much parallelism of which a program may take advantage on a more realistic finite-processor parallel machine.

Appendix A: Jimple Grammar

Detailed Grammar for Jimple

The following grammar is for the Jimple intermediate language used by the static analysis in this thesis via the Soot Java optimization framework. This representation is reminiscent of Java syntax, but it breaks down more complex statements containing multiple expressions into simpler statements, as described below. This grammar captures the details that could be relevant for static analysis and instrumentation of Java programs. This section was compiled by Prof. Rountev.

Consider the following grammar with starting non-terminal $\langle Program \rangle$.

$$\langle Program \rangle ::= \langle Class \rangle^+$$
$$\langle Class \rangle \quad ::= \langle Field \rangle^* \ \langle Method \rangle^*$$
$$\langle Method \rangle \quad ::= \langle Statement \rangle^*$$

The nonterminal $\langle Class \rangle$ represents both classes and interfaces; for the rest of this document, the term "class" will be used to refer to both classes and interfaces. The program contains at least one class. Certain classes are designated as *library classes*, as they belong to library classes. The rest of the classes are *client classes*. If we consider analysis of a complete program, one of the client classes is designated as the *main class* — it contains the main method of the program. In addition to the set of classes, the program defines a *class hierarchy*: a DAG which captures the `extends` and `implements` relationships.

A class contains some combination of fields and methods. Each field has a name, a type (primitive type or reference type), and modifiers which are one or more of the following: `public`, `protected`, `private`, `static`, `final`, `transient`, `volatile`. Each method has a name, a sequence of parameter types, a return type, and modifiers which are one or more of the following: `public`, `protected`, `private`, `abstract`, `static`, `final`, `synchronized`, `native`, `strictfp`. Some of the methods in the class can be designated as *constructors*; all and only such methods have names `<init>`. Every class has at least one constructor; every interface has zero constructors. A class has zero or one *static initializer* methods, with name `<clinit>`, with a `static` modifier.

A method has a body containing an *ordered sequence* of zero or more statements. All and only abstract and native methods have zero statements. The different categories of statements are defined as follows:

⟨*Statement*⟩ ::= ⟨*ReturnStmt*⟩
     | ⟨*ReturnVoidStmt*⟩
     | ⟨*ThrowStmt*⟩
     | ⟨*GotoStmt*⟩
     | ⟨*IfStmt*⟩
     | ⟨*SwitchStmt*⟩
     | ⟨*MonitorStmt*⟩
     | ⟨*InvokeStmt*⟩
     | ⟨*IdentityStmt*⟩
     | ⟨*AssignStmt*⟩

The details of individual categories of statements are as follows:

⟨*ReturnVoidStmt*⟩ ::= `return`
     // return nothing
⟨*ReturnStmt*⟩   ::= `return` ⟨*SimpleValue*⟩
     // return a constant or the value of a local variable
⟨*ThrowStmt*⟩   ::= `throw` ⟨*Local*⟩
     // throw the exception pointed-to by the local variable
⟨*GotoStmt*⟩   ::= `goto` `stmt_reference`
     // jump to a statement in the same method

$\langle IfStmt \rangle$ ::= `if` $\langle ConditionExpr \rangle$ `stmt_reference`
         // conditional jump to a statement in the same method

$\langle SwitchStmt \rangle$ ::= `switch` $\langle Local \rangle$ `stmt_reference`$^+$
         // conditional jump to a statement in the same method

$\langle MonitorStmt \rangle$ ::= `monitor` $\langle Local \rangle$
         // enter or exit a `synchronized` block

$\langle InvokeStmt \rangle$ ::= $\langle InvokeExpr \rangle$
         // call without return value

$\langle IdentityStmt \rangle$ ::= $\langle Local \rangle$ `=` $\langle IdentityRef \rangle$
         // parameter passing or exception catching

The `stmt_reference` above is a pointer to some other statement in the same method body. A few of the non-terminals used above are as follows:

$\langle Constant \rangle$ ::= $\langle NumericConstant \rangle$
         $\mid \langle StringConstant \rangle \mid \langle NullConstant \rangle \mid \langle ClassConstant \rangle$

$\langle Local \rangle$ ::= `id` // variable name

$\langle SimpleValue \rangle$ ::= $\langle Local \rangle \mid \langle Constant \rangle$

$\langle IdentityRef \rangle$ ::= $\langle ThisRef \rangle \mid \langle ParameterRef \rangle \mid \langle CaughtExceptionRef \rangle$

$\langle ThisRef \rangle$ ::= `this` // for instance methods

$\langle ParameterRef \rangle$ ::= `formal`$_i$
         // i-th formal parameter of the method, with $i \geq 1$

$\langle CaughtExceptionRef \rangle$::= `caughtexception`
         // artificial variable pointing to exception at a `catch` clause

There are two kinds of $\langle IdentityStmt \rangle$. First, when the right-hand side is a formal parameter (including the implicit formal parameter `this`), this statement appears at the very beginning of the method. This is the only place where the formals appear in the body. The second kind of $\langle IdentityStmt \rangle$ is used to represent the effect `catch (Exception e)`: the `caughtexception` points to the exception object upon entry into the `catch`, and its value is used to set the value of local `e`.

$\langle AssignStmt \rangle$ ::= $\langle Local \rangle$ `=` $\langle GeneralRhs \rangle$
           $\mid$ $\langle ArrayRef \rangle$ `=` $\langle SimpleValue \rangle$
           $\mid$ $\langle FieldRef \rangle$ `=` $\langle SimpleValue \rangle$

$\langle ArrayRef \rangle$ ::= $\langle Local \rangle$ `[` $\langle SimpleValue \rangle$ `]`

$\langle FieldRef \rangle$ ::= $\langle StaticFieldRef \rangle \mid \langle InstanceFieldRef \rangle$

$\langle StaticFieldRef \rangle$ ::= `id`

$\langle InstanceFieldRef \rangle$::= $\langle Local \rangle$ `.` `id`

The general form of the right-hand side of an ⟨*AssignStmt*⟩ is this:

⟨*GeneralRhs*⟩	::=	⟨*SimpleValue*⟩
	|	⟨*AnyNewExpr*⟩ // creation of an object or an array
	|	⟨*FieldRef*⟩
	|	⟨*ArrayRef*⟩
	|	⟨*InvokeExpr*⟩ // method call with a return value
	|	⟨*BinopExpr*⟩ // binary operation
	|	⟨*UnopExpr*⟩ // unary operation
	|	⟨*CastExpr*⟩ // casting
	|	⟨*InstanceofExpr*⟩ // x `instanceof` C
⟨*AnyNewExpr*⟩	::=	⟨*NewExpr*⟩ | ⟨*NewArrayExpr*⟩ | ⟨*NewMultiArrayExpr*⟩
⟨*NewExpr*⟩	::=	`new` ⟨*Type*⟩
		// create an instance of the corresponding class
⟨*NewArrayExpr*⟩	::=	`newarray` ⟨*Type*⟩ `of length` ⟨*SimpleValue*⟩
		// ⟨*Type*⟩ is the component type (could be an array type)
⟨*NewMultiArrayExpr*⟩	::=	`newmultiarray` ⟨*Type*⟩ `of length` ⟨*LengthList*⟩
		// ⟨*Type*⟩ is the component type (could be an array type)
⟨*LengthList*⟩	::=	⟨*SimpleValue*⟩
	|	⟨*SimpleValue*⟩ ⟨*LengthList*⟩
		// lengths in all allocated dimensions
⟨*BinopExpr*⟩	::=	⟨*SimpleValue*⟩ `binop` ⟨*SimpleValue*⟩
		// `binop` is some binary operator
⟨*UnopExpr*⟩	::=	`unop` ⟨*SimpleValue*⟩ // `unop` is some unary operator
⟨*CastExpr*⟩	::=	`cast` ⟨*SimpleValue*⟩ `to` ⟨*Type*⟩
⟨*InstanceofExpr*⟩	::=	⟨*Local*⟩ `instanceof` ⟨*Type*⟩
		// the type is a reference type

The call expressions have the following form:

⟨*InvokeExpr*⟩	::=	⟨*StaticInvokeExpr*⟩ // call to a static method
	|	⟨*InstanceInvokeExpr*⟩ // call to an instance method
⟨*StaticInvokeExpr*⟩	::=	`id (` ⟨*SimpleValue*⟩* `)`
		// zero or more simple values as actual parameters
⟨*InstanceInvokeExpr*⟩	::=	⟨*SpecialInvokeExpr*⟩ // call without dynamic dispatch
	|	⟨*VirtualInvokeExpr*⟩ // call with dynamic dispatch
⟨*SpecialInvokeExpr*⟩	::=	⟨*Local*⟩ `.` `id (` ⟨*SimpleValue*⟩* `)`
⟨*VirtualInvokeExpr*⟩	::=	⟨*Local*⟩ `.` `id (` ⟨*SimpleValue*⟩* `)`
		// the method is a compile-time target

Bibliography

[1] T. Austin and G. Sohi. Dynamic dependency analysis of ordinary programs. In *International Symposium on Computer Architecture*, pages 342–351, 1992.

[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, 2006.

[3] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[4] M. Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computers*, 37(9):1088–1098, Sept. 1988.

[5] M. Lam and R. Wilson. Limits of control flow on parallelism. In *International Symposium on Computer Architecture*, pages 46–57, 1992.

[6] J. Larus. Loop-level parallelism in numeric and symbolic programs. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):812–826, July 1993.

[7] J. Mak and A. Mycroft. Limits of parallelism using dynamic dependence graphs. In *International Workshop on Dynamic Analysis*, pages 42–48, 2009.

[8] M. Marron, M. Mendez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur. Sharing analysis of arrays, collections, and recursive structures. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Egineering*, pages 43–49, 2008.

[9] M. Postiff, D. Greene, G. Tyson, and T. Mudge. The limits of instructions level parallelism in SPEC95 applications. In *ASPLOS Workshop on Interaction Between Compilers and Computer Architecture*, 1998.

[10] L. Rauchwerger, P. Dubey, and R. Nair. Measuring limits of parallelism and characterizing its vulnerability to resource constraints. In *International Symposium on Microarchitecture*, pages 105–117, 1993.

[11] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction*, LNCS 1781, pages 18–34, 2000.

[12] D. Wall. Limits of instruction-level parallelism. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, 1991.