An Aspect Pointcut for Parallelizable Loops

by

John S. Dean

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
Computer Science

Graduate School of Computer and Information Sciences
Nova Southeastern University

2013

We hereby certify that this dissertation, submitted by John Dean, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

_____ _____

Francisco J. Mitropoulos, Ph.D. Date
Chairperson of Dissertation Committee

_____ _____

Michael J. Laszlo, Ph.D. Date
Dissertation Committee Member

_____ _____

Junping Sun, Ph.D. Date
Dissertation Committee Member

Approved:

_____ _____

Eric S. Ackerman, Ph.D. Date
Dean, Graduate School of Computer and Information Sciences

Graduate School of Computer and Information Sciences
Nova Southeastern University

2013

An Abstract of a Dissertation Submitted to Nova Southeastern University in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

An Aspect Pointcut for Parallelizable Loops

by
John S. Dean

July 2013

This study investigated the need for a pointcut for parallelizable loops in an aspect-oriented programming environment. Several prototype solutions exist for loop pointcuts, but the solutions are not very granular. In particular, they are not able to differentiate between loops that are parallelizable and those that are not. Being able to identify parallelizable loops automatically, as part of an aspect-oriented compiler's weaving process, is particularly important because (1) manually identifying parallelizable loops is known to be a difficult problem and (2) aspectizing parallelized loops can lead to a reduction in code tangling and an increase in separation of concerns.

This paper describes the concepts behind the loop-pointcut problem. It then describes the approach used in this study for implementing a solution in the form of an aspect-oriented Java compiler with a parallelizable loop pointcut. Identifying parallelizable loops is known to be a difficult problem, and as such, this study's parallelizable loop pointcut implements a heuristic solution. The pointcut identifies many parallelizable loops as being parallelizable, but in erring on the side of conservatism, there are some parallelizable loops that the pointcut is unable to identify as parallelizable.

To test the parallelizable-loop pointcut, the pointcut was applied to a benchmark set of parallelizable programs. There were two versions of each benchmark program – (1) an aspect-oriented version, where the aspect-oriented compiler's weaver added the multi-threading functionality, and (2) a non-aspect-oriented version, where the benchmark program's source code directly implemented the multi-threading functionality. For each benchmark program, the output from the aspect-oriented version was compared to the output from the non-aspect-oriented version. The study found that each loop that was deemed parallelizable by the aspect-oriented benchmark program was executed in parallel (with multiple threads) by both versions of the program – the aspect-oriented version and the non-aspect-oriented version. There were some loops in the non-aspect-oriented benchmark programs that were deemed parallelizable and executed in parallel, but those same loops were deemed non-parallelizable by their associated aspect-oriented benchmark program. This discrepancy is explained by the study's conservative approach to identifying loops as parallelizable.

Acknowledgments

I would like to thank my Ph.D. advisor, Dr. Frank J. Mitropoulos for his initial encouragement to pursue my interest in aspect-oriented loop pointcuts and his support, advice, and patience throughout the past five years. He kept me going through the tough times and without his backing, this would not have been possible. I would also like to thank my examining committee members Dr. Michael J. Laszlo and Dr. Junping Sun, whose support in the dissertation process and in my course work was very much appreciated.

Thanks also to Dr. Ron Krawitz and the rest of the gang in the Nova SCIS Dissertation Support Group. Although I wasn't as active in the group as others, I enjoyed reading the posts and knowing that I wasn't alone.

I am indebted to my family, for their unwavering encouragement and support.

I planned to use an existing aspect-oriented loop pointcut as the starting point for my parallelizable loop pointcut. Although the loop pointcut that I planned to use was presented in a paper at an international conference, and the paper is cited by many other published articles, I was unable to get that loop pointcut to work. On the abc Users' Group Forum, I asked if anyone had heard of an alternative study that implemented an aspect-oriented loop pointcut. Dr. Seyed-Hossein Sadat-Kooch-Mohtasham (Hossein) replied that his transcut software tool in his just-completed Ph.D. dissertation might be able to help. Soon thereafter, I was able to get his transcut pointcut to work, and it did indeed prove to be a good starting point for my parallelizable loop pointcut. In my implementation efforts, if a task required me to understand something about his code that I struggled with, I asked Hosssein, and he provided thoughtful and detailed replies. Digging back into such a large and complex software tool is a very difficult undertaking. I was inspired by his altruism. He is one of the kindest people I have ever met. In the future, I hope that I will be able to pay his goodwill forward to some other doctoral student.

Table of Contents

List of Figures

Figures

Chapter 1

Introduction

Background

This paper describes the need for an aspect pointcut for loops that can be safely parallelized. Having such a pointcut should enable programmers to write programs that are relatively elegant and relatively efficient. The elegance can be achieved through improved separation-of-concerns and low coupling characteristics that are common in aspect-oriented programs. The efficiency can be achieved through the parallelization of loops that are found to be parallelizable.

This Introduction chapter provides the study's problem statement, goal, and significance. In addition, the Introduction chapter presents barriers and issues, the research question addressed by the study, and limitations and delimitations of the study. Next, the Review of the Literature chapter provides background information that sets the context for the study. In particular, the literature review chapter presents concepts and significant findings of relevant prior studies. Next, the Methodology chapter describes the manner in which the study was conducted. In doing so, it describes techniques from prior studies that the study drew from, presents the steps taken in conducting the study, and lists resource requirements. The Results chapter provides the study's outcomes. In particular, the design and implementation of a parallelizable loop detection algorithm was one of the significant outcomes of this study, and, as such, the Results chapter provides an in-depth explanation of the parallelizable loop detection algorithm. In addition, the

chapter provides results from the study's tests that illustrate the success in implementing a loop pointcut as part of an aspect-oriented compiler. Finally, the Conclusions, Implications, Recommendations, and Summary chapter ascertains whether the study's goals have been met, discusses the impact of the study's results on future research, and makes recommendations for related future research.

Problem Statement

Loops sometimes contain operations that can be executed in a parallel manner via a multitasking environment or a multiprocessing environment. When loops with parallelizable operations are executed in parallel, such parallelization can lead to improved performance (Harbulot & Gurd, 2004, p. 323). However, traditional loop parallelization techniques (adding thread code directly to the original program code), can create substantial code tangling between the program's original primary concern(s) and the newly introduced parallelization concern (Harbulot & Gurd, 2004). Tangled code leads to programs that lack modularity and cohesion. As such, tangled-code programs tend to be hard to understand and develop. Also, they tend to be hard to maintain because changing the functionality of tangled programs requires developers to go through the process of mentally untangling and then retangling the code (Kiczales et al., 1997). Furthermore, tangled-code modules tend to be hard to reuse since they don't focus on just one task (Laddad, 2003). Using aspects for parallelizable loops (with the introduction of a parallelizable loop pointcut) can help to untangle the code by separating the loops' core concerns from the loops' cross-cutting parallelizing concern (Harbulot & Gurd, 2004).

Such separation can lead to programs that are easier to maintain and program modules that are easier to reuse (Laddad, 2003).

Dissertation Goal

One goal of this study was to define a pointcut for loops that are safely parallelizable. To achieve this goal, the study formalized the constraints involved in classifying a loop as parallelizable. Another goal of this study (the primary goal) was to implement the defined parallelizable loop pointcut. To achieve this goal, the researcher designed and implemented an algorithm that determines whether a given loop is parallelizable. Determining whether a loop is parallelizable is known to be a very difficult problem (Aho, Lam, Sethi, & Ullman, 2007; Kyriakopoulos & Psarris, 2004), so much of this study's effort focused on that issue. The final goal of this study was to modify an existing aspect-oriented compiler so that its matching and weaving mechanisms worked with the new parallelizable loop pointcut.

Research Question and Hypothesis

With the primary goal of this study being the implementation of a parallelizable loop pointcut, the study's primary question was whether the parallelizable loop pointcut could be implemented correctly. The study's researcher hypothesized that it could be implemented correctly. To test the hypothesis, the study used the parallelizable loop pointcut to apply multi-threading advice to a benchmark set of programs that were known to be parallelizable. There were two versions of each benchmark program – (1) an aspect-oriented version, where the aspect-oriented compiler's weaver added the multi-threading

functionality, and (2) a non-aspect-oriented version, where the benchmark program's source code directly implemented the multi-threading functionality. The study added print statements to the benchmark programs to determine whether the two versions for each benchmark program produced the same output in terms of the program's calculations (e.g., matrix multiplication) and also in terms of the number of threads used to execute each of the parallelizable loops. In comparing the outputs, the study showed that a parallelizable loop pointcut could be implemented correctly. More specifically, it showed that an aspect-oriented compiler with a parallelizable loop pointcut could detect parallelizable loops and apply the multi-threading advice correctly.

Relevance and Significance

In the past, prototype solutions for loop pointcuts have been relatively coarse, with each prototype implementing just one loop pointcut construct (Eaddy & Aho, 2006; Harbulot & Gurd, 2006; Rajan & Sullivan, 2005; Rho, Kniesel, & Appeltauer, 2006). In reviewing the relevant research literature, no prior studies were found that implemented a loop pointcut that verified whether a matched loop join point was parallelizable. By implementing that verification process, this study's pointcut can prevent inappropriate attempts to parallelize code that is inherently non-parallelizable.

Being able to aspectize the parallelization of loops is particularly important because loop parallelization code is a heterogeneous concern, and heterogeneous concerns tend to have tangled code. A heterogeneous concern is a concern in which the code is present in multiple places and the context code (the code in which the heterogeneous code is embedded) is different (Trifu & Kuttruff, 2005). By aspectizing the parallelization of

loops, this study's parallelizable loop pointcut has made it easier to reduce code tangling with loop parallelization code. And a reduction in code tangling can lead to improvements in maintenance, debugging, and code reuse (Eaddy & Aho, 2006).

By implementing an aspect pointcut that targets parallelizable loops, this study has made it easier for programmers to introduce safe parallelization to their programs, and therefore should make it more likely that such parallelization occurs. The benefit of such parallelization is made clear by *Amdahl's Law*, which states that the speedup of a parallelized program is a function of *f*, the fraction of the program's executed code that is parallelized, and *p*, the number of processors employed to run the program (Aho et al., 2007):

$$\frac{1}{(1-f)+(f/p)}$$

So, for example, if 80% of a program's executed code is parallelized, and 10 processors are used, then the speedup is approximately 3.57.

This study's focus on speeding up the execution of loops has been deemed to be a worthwhile goal. According to (Aho et al., 2007), programs typically spend most of their time executing loops, so improving the performance of loops can have a significant impact on the overall performance of programs.

Barriers and Issues

Why Have this Study's Goals Not Been Met in the Past?

Aspect-oriented programming is a relatively new technology, having originated in 1997 (Kiczales et al., 1997). As such, aspect pointcuts have been developed for just a

subset of Java's constructs and there is a need for more and finer-grained aspects (Eaddy & Aho, 2006; Harbulot & Gurd, 2006; Rho et al., 2006).

This study's focal construct, the loop, is more difficult to map to a pointcut(s) than other constructs because loops don't have standard arguments (like method arguments) or identifiers that can be used to identify them (Harbulot & Gurd, 2006). Labels cannot be used effectively to identify loops because programmers use labels infrequently. Additionally, labels could not have been used to identify loops in this study because labels appear only in source code, not in a program's resulting generated bytecode, and this study's aspect-oriented compiler identified pointcut join points at the bytecode level.

If pointcut join points were identified at the source code level, then there would be a need for different pointcuts for while, do, and for loops. In that case, programmers' personal preferences for loop types would impact which pointcuts were used, and the types of pointcuts used could impact the effectiveness of a particular program's aspectization (Harbulot & Gurd, 2006). To avoid such programmer-specific variability, this study identified pointcut join points at the bytecode level, rather than at the source code level. That identification process required an understanding of bytecode. That increased this study's challenge since an understanding of bytecode is less common than an understanding of Java source code.

Degree of Difficulty of this Study's Solution

This study's literature review found only a few studies that implemented loop pointcuts that matched at the bytecode level. One such example was Harbulot and Gurd's LoopsAJ aspect-oriented compiler. It was particularly appealing as a possible starting

point for this study because LoopsAJ differentiates between loops with different numbers of exit points, and that differentiation is known to be helpful when identifying loops that are parallelizable (Bik & Gannon, 1997; Harbulot & Gurd, 2006). Unfortunately, as this dissertation effort progressed, it became clear that the LoopsAJ code was difficult to work with. In an email correspondence, Dr. Harbulot said he was "sorry the code is not of better quality," and he acknowledged that his prototype's code "could have been written more cleanly" (personal communication, June 22, 2009).

This study's primary challenge was the determination of whether a given loop was parallelizable. A parallelizable loop is a loop which can have its iterations executed in any order on different processors. Determining the parallelizability of a loop is known to be a difficult problem, particularly when array references and nested loops are involved (Blume et al., 1994; Kyriakopoulos & Psarris, 2004). Most studies that attempt to detect loop parallelizability engage in data dependency analysis. Data dependency analysis, described in detail later on, can be quite complex, and NP-complete in certain circumstances (Aho et al., 2007; Kyriakopoulos & Psarris, 2004).

To avoid having to "reinvent the wheel," this study's researcher initially planned to borrow from the ReLooper tool, which, according to (Dig, Tarce, Radoi, Minea, & Johnson, 2009), was supposed to determine whether a loop can be safely executed in parallel. Unfortunately, as this dissertation effort progressed, it became clear that the ReLooper code was difficult to work with. Dr. Dig, one of ReLooper's chief architects, said "We are now at the third re-implementation of the program analysis for detecting races in loops over arrays. This is a very hard problem to solve statically. I know that our third attempt is not ready for a release" (personal communication, May 31, 2011). In a

subsequent email, Dr. Radoi, another one of ReLooper's chief architects, stated that ReLooper was being rewritten, this time using Scala instead of Java (personal communication, February 14, 2012). The purpose of this study was to implement a parallelizable loop pointcut as part of an aspect-oriented compiler. Since the most popular and extensible aspect-oriented compilers are written in Java, it would have been difficult to use ReLooper's Scala code within the context of this study.

Limitations and Delimitations

A limitation is something that might affect the study, but it is not under the researcher's control. As noted above, this study implemented a parallelizable loop pointcut and used it to apply multi-threading advice to a benchmark set of programs that were known to be parallelizable. By the nature of benchmarks, the benchmark programs' subject areas and algorithms were out of the control of the researcher. As such, they were considered a limitation of the study. The researcher attempted to choose benchmark programs that lent themselves equally well to manual and aspectized loop parallelization, but some of the benchmark programs did not lend themselves equally well to aspectization. Consequently, the study's parallelizable loop pointcut was unable to detect some of the benchmark program loops as parallelizable, even though they were shown to be parallelizable in the non-aspect-oriented versions of the benchmark programs, where the source code directly implemented the multi-threading functionality. Based on Zhong's findings, this result was not surprising (Zhong, Mehrara, Lieberman, & Mahlke, 2008). Zhong found that parallelizable-loop detection tools were less effective at identifying parallelizable loops than manual analysis.

Another limitation of the study is the nature of the parallelizable loop detection mechanism in terms of static versus dynamic. Since aspect-oriented compilers use static techniques to match pointcuts, and this study relied on an existing aspect-oriented compiler as a starting point, this study's solution necessarily relied on static techniques for matching its new parallelizable loop pointcut.

A delimitation is something that might affect the study, and it is under the researcher's control. As noted above, most studies that attempt to detect loop parallelizability engage in data dependency analysis, and data dependency analysis can be quite complex, and NP-complete in certain circumstances. With such a difficult problem, this study designed and implemented a heuristic solution – a solution that identifies some parallelizable loops as parallelizable, but not all such loops. The aggressiveness of the heuristic solution was in in the control of the researcher, where "aggressiveness" refers to the effort in identifying a higher percentage of parallelizable loops as parallelizable. With such parallelizable-loop detection aggressiveness in the control of the researcher, it is considered to be a delimitation of the study.

Definitions of Terms

AspectBench Compiler (abc) – an alternative compiler to AspectJ's original compiler, AspectJ Compiler (ajc). abc was specifically designed to be extensible.

back edge – An edge from y to x is a back edge if x dominates y.

code tangling – When a module incorporates multiple concerns such that the concerns are intermixed within the module.

concern – A functionally cohesive unit that must be implemented as part of a program's overall goal.

control-flow graph – An illustration of a program's possible execution paths, using nodes for program instructions and edges for connections between instructions. A connection arrow goes from node A to node B if A's instruction might be executed immediately before node B's instruction.

cross-iteration dependence – When the execution of a statement in one iteration of a loop is dependent on the execution of a statement in another iteration of the loop.

dependence (in a program dependency graph) – A node B is dependent on node A if the only way to reach B from the program's starting point is by going through A.

dominator – A node x is a dominator of node y (i.e., it *dominates* node y) if for every path from the starting node to node y, the path goes through node x.

Jimple – An intermediate representation of Java code between source code and bytecode. In this study, the Jimple application programming interface (API) was used extensively to implement the analysis of Java constructs.

join point shadow (in a program dependency graph) – A subset of nodes in a PDG region such that the nodes form a single-entry-single-exit sequence of nodes in the PDG's corresponding control-flow graph.

natural loop – A natural loop for back edge y to x is the set of nodes along a path from x to y, excluding the paths that revisit node x.

parallelizable loop – A loop which can have its iterations executed in any order on different processors such that the loop's functionality is unaffected.

program dependency graph (PDG) – An illustration of a program's control dependencies, using nodes for program instructions and regions and edges for connections between them. A connection arrow goes from node A to node B if node B is dependent on node A. This definition is for PDGs used in this study. Other PDGs exist which have more features.

region (in a program dependency graph) – A region is a set of nodes in a PDG such that either all of the region's nodes are executed or none of them are executed.

separation of concerns – When a program's functional behaviors are implemented in separate modules with minimal overlap between the functional behaviors.

shadow matching – The process of matching pointcuts in an aspect file with join points in a program that is being aspectized.

shared object – An object that is accessed in different iterations of the same parallel loop.

Soot – A software framework that helps to analyze, manipulate, and optimize Java code. In this study, Soot's application programming interface (API) was used extensively to implement the analysis of Java constructs and the analysis of nodes in a program's program dependency graph.

transcut – A pointcut that contains a group of other pointcuts, such that the `transcut` pointcut matches a join point if the transcut's contained pointcuts match with join points inside the transcut join point. The contained pointcuts and contained join points must be in the same order, but they do not have to be contiguous.

worker object creation pattern – A design pattern where a method call is replaced with code that (1) instantiates an object that contains the method as one of the object's members, and (2) calls the object's method.

Summary

This Introduction chapter has provided an overview of the study's primary goal – implementing a pointcut for parallelizable loops for an aspect-oriented compiler. The introduction presented the need for such a pointcut and the significance of implementing such a pointcut. Next, the introduction discussed barriers to the success of this study and limitations and delimitations of the study. Finally, this Introduction chapter presented definitions of terms that are used throughout the remainder of this paper.

Chapter 2

Review of the Literature

Introduction

The purpose of this Review of the Literature chapter is to provide background information intended to help the reader understand the basic concepts and needs in the study's domain, which is aspect pointcuts for parallelizable loops. It presents significant findings of relevant prior studies. The chapter begins with a description of the general need for more finely grained pointcuts in aspect-oriented compilers. Next, loop types and loop pointcuts are discussed in the context of one of the prior studies upon which this study is based – Harbulot and Gurd's LoopsAJ study (2006).

The most challenging portion of this study was the detection of parallelizable loops. With that in mind, the literature review first presents a discussion of loop parallelizability in general. It then overviews various techniques that prior studies have used for parallelizing loops. It then presents various techniques that prior studies have used for the detection of loops that are safely parallelizable.

Fine-Grained Pointcut Code Coverage

Most aspect-oriented compilers are limited in their coverage of the language that they aspectize. Various independent studies have indicated the need for greater coverage through an increased number of pointcuts (Rajan & Sullivan, 2005; Rho et al., 2006). For example, Kniesel and Austermann (2002) found that to achieve 100% aspectized code

coverage of a program for quality assurance purposes (as required by a customer contract noted in their study), it would be necessary to have access to every statement in the program. Such comprehensive access would include pointcuts for loops. Several studies have provided prototype solutions for loop pointcuts, but their resulting loop pointcuts are somewhat general in nature – they identify a loop, but they don't attempt to detect whether the loop is parallelizable (Eaddy & Aho, 2006; Harbulot & Gurd, 2006; Rajan & Sullivan, 2005; Rho et al., 2006). Providing a parallelizable loop pointcut would not only further the goal of having more fine-grained pointcut code coverage, but it would also help programs to receive the benefit of parallelized loops, as discussed earlier in this paper.

Loop Types and Loop Pointcuts

Harbulot and Gurd (2006) found that three basic types of loops exist, each with their own aspectization capabilities. The three types are classified according to a loop's number of exit nodes and number of successor nodes. An exit node corresponds to a statement within a loop that causes the loop to terminate. A successor node corresponds to a statement that can possibly execute immediately after the loop's termination. The three loop types, as shown in Figure 1, are: (1) a loop with one exit node and one successor node, (2) a loop with more than one exit node and one successor node, and (3) a loop with more than one successor node.

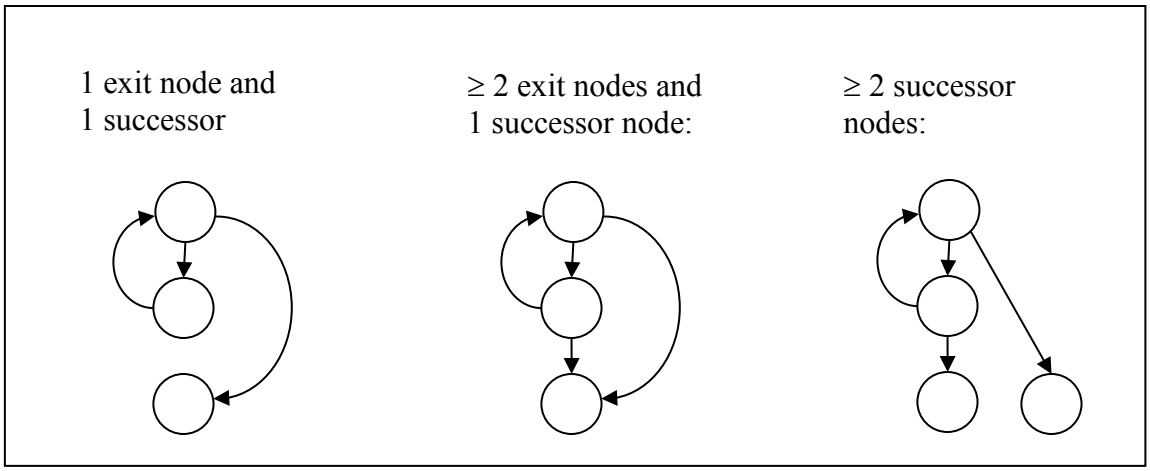

Figure 1. Three types of loops

Harbulot and Gurd (2006) implemented a pointcut for just the first type of loop shown in Figure 1. They chose to implement that type of loop, with its one-exit-node, one-successor-node structure, because it has the most potential for aspectization functionality. Specifically, it can handle before, after, and around aspect advice, and it can provide context exposure. Context exposure means that a loop's pointcut arguments are accessible and accurate. A loop's pointcut arguments consist of min, max, and step, which correspond to the three components in a `for` loop's heading. For example, the min, max, and step values for the following for loop header are 0, 10, and 1, respectively:

```
for (int i=0; i<10; i++)
```

In the above code, the reason for min being 0 and max being 10 should be self-evident. The reason that step is 1 is because the index variable i increments by 1 at the end of each loop iteration.

If a loop contained a `break` statement, the `break` statement would represent an exit node and cause the loop to fall into the loop type characterized by having more than one exit node and one successor node. If a pointcut were developed for that type of loop, it would lend itself to before, after, and around advice, but not context exposure. The

`break` statement would cause the loop's iteration space (the set of iterations that the loop executes) to become unknown. That would cause the loop's max argument to become the largest possible upper limit to the loop's number of iterations rather than the actual upper limit to the loop's number of iterations. That uncertainty would lead to a lack of context exposure (Harbulot & Gurd, 2006).

If a loop had more than one successor node, then there would be multiple places where after advice would have to go. If the aspect compiler handled multiple weaving points, then after advice would be possible. With multiple successor nodes, the end of the loop would be unclear. Consequently, around advice wouldn't work. Multiple successor nodes imply that a loop has multiple exit nodes, and, as explained above, multiple exit nodes lead to a lack of context exposure (Harbulot & Gurd, 2006). Note Figure 2, a modified version of (Harbulot & Gurd, 2006, p. 69), which shows the aspect-oriented characteristics of the three types of loops.

	Before	After	Around	Context exposure
1 exit node, 1 successor node	√	√	√	√
Multiple exit nodes, 1 successor node	√	√	√	x
Multiple successor nodes	√	√, (if multiple weaving points)	x	x

Figure 2. Different loop types and their weaving capabilities

Although Harbulot and Gurd (2006) did not implement pointcuts for the bottom two types of loops shown in Figure 2, they recognized that having a pointcut for the second type of loop could be useful. Specifically, they recognized that a loop with multiple exit

nodes and one successor node could have around advice applied to it. However, loops with multiple exit nodes are generally considered to be unsafe candidates for parallelization (Aho et al., 2007). That should make sense because with more than one exit point, one of the exit points would normally be based on a condition within the body of the loop. If the loop were to be parallelized, then presumably one of the processor's loop iterations would find a true condition and want to exit the loop. But to exit the loop and preserve the original program's integrity, the program would have to (1) immediately stop the other processors' loop iterations, and (2) check to make sure that the other processors' loop iterations hadn't executed code that should not have been executed prior to the exit condition becoming true, and undo those executions if appropriate. The second case is impractical because the effort involved in implementing it (if that's even possible) would counteract the speedup benefit due to the parallelization. Since loop parallelization is the focus of this study, there is no need for this study to consider Harbulot and Gurd's non-parallelizable second type of loop.

Harbulot and Gurd recognized that weaving around advice into their third type of loop, with multiple exits and multiple successors, would not be possible, even theoretically (2006). Since around advice is required for loop parallelization, Harbulot and Gurd's third type of loop is not parallelizable. Since loop parallelization is the focus of this study, there is no need for this study to consider Harbulot and Gurd's non-parallelizable third type of loop.

In implementing a loop pointcut for their aspect-oriented compiler, Harbulot and Gurd (2006) did not attempt to check the parallelizability of matching join point loops. Checking a loop to determine whether it can be parallelized is a complex problem

(Kyriakopoulos & Psarris, 2004). With this study's focus on loop parallelization, this study did include such verification for its loop pointcut. The primary means for detecting loop parallelizability is to check for the lack of statement executions in one iteration of a loop that are dependent on statement executions in another iteration of the loop (Blume et al., 1994).

LoopsAJ

Some aspect compilers perform their weaving operations at the source code level, while some perform their weaving operations at the bytecode level (Harbulot & Gurd, 2006). This study's resulting compiler wove its loop advice at the bytecode level. This coincides with Harbulot and Gurd's LoopsAJ compiler and the de facto standard aspect-oriented compiler, AspectJ (Hilsdale & Hugunin, 2004). Weaving at the bytecode level has several advantages. Java compilers are able to recognize different forms of loops (e.g., `for` loops, `while` loops, and sequential branching that jumps to prior code) and produce generic loop bytecode. Having generic bytecode as a starting point for an aspect compiler makes it easier for the aspect compiler to process loops. If source code were used as a starting point, then the programmer would need to either (1) first refactor the source code to produce a common loop structure for aspectization purposes, or (2) allow multiple source code loop structures (e.g., `for` loop, `while` loop) to be aspectized. If multiple source code loop structures were aspectized, then the resulting aspectized code would reflect the different styles exhibited by the source code programmers, and that would lead to less robust solutions (Harbulot & Gurd, 2006).

Most AspectJ pointcuts rely on programmer-defined names to select particular join points (Harbulot & Gurd, 2006). For example, the following AspectJ pointcut relies on the programmer-defined names `Employee` and `set` to select all the one-parameter `public set` methods in the `Employee` class :

```
public void Employee.set*(*)
```

Loop join points are harder to select because loops don't have names. Some programmers use loop labels, which can partially substitute for loop names, but loop labels are not included in bytecode, and AspectJ, LoopsAJ and this study's aspect compiler all rely on bytecode as their input source (Harbulot & Gurd, 2006).

Harbulot and Gurd's solution for selecting loop join points is to rely on loops' exposed context (i.e., the quantity and types of loop arguments). For example, the following LoopsAJ advice uses the `loop` and `args` pointcuts to select loops with integer min, max, and step values and a `double` array whose elements are looped through.

```
void around(int min, int max, int step, double[] x):
  loop() && args(min , max , step , x) &&
  within(Employee.adjustSalary(..))
{ ...
```

Note that the above `within` construct limits the loop selection to loops that are within `Employee`'s `adjustSalary` method.

Loop Parallelizability

A loop is *parallelizable* if iterations of its body can be executed on different processors without customized synchronization code and without compromising the loop's original functionality (Psarris & Kyriakopoulos, 2003). For several reasons, loops tend to be better candidates for parallelization than sequential statements. When

statements are parallelized, their order of execution cannot be assumed. Usually, the order of statements within a sequential block tends to be critical for the correctness of the block's execution. On the other hand, it is fairly common that different iterations of a loop body can be executed in different orders with no adverse effect on the correctness of the loop's execution (Aho et al., 2007). Another parallelization advantage of loops over sequential statements is that each loop iteration's execution requires approximately the same processing power (because each iteration processes the same loop body code), and that makes it relatively easy to keep processors occupied continuously in a parallel fashion (Aho et al., 2007).

A disadvantage of parallelizing sequential statements is that the parallelization benefit is limited to the (fixed) number of statements in the sequence. On the other hand, if a loop body can be parallelized, the amount of parallelization scales with the size of the data (assuming that the loop processes data). As the amount of data increases, the number of loop iterations increases. As the number of loop iterations increases, the fraction of executed parallelized code increases. Amdahl's Law tells us that as the fraction of executed parallelized code increases, program speedup will occur as a function of the number of parallel processors (Aho et al., 2007).

How to Parallelize Loops

Besides using a parallelizable loop pointcut with aspects (as done by this study), there are two other basic solutions for parallelizing loops – (1) explicit parallelization using Java threads and (2) calls to external library functions (Harbulot & Gurd, 2004).

The first of the two "other basic solutions" (explicit parallelization) is where the programmer implements parallelization with source code. C can spawn multiple parallel thread processes directly, while Java can instantiate and execute threads which run asynchronously on multiple processors. An example of explicit parallelization with Java threads is presented in Figure 3, which comes from (Harbulot & Gurd, 2004, p. 124). The figure shows how a new thread is instantiated by calling the `Thread` constructor, how it is executed by calling the `start` method, and how it is waited on for completion by calling the `join` method.

```
public class Example
{
  ...
  public void sequentialExample()
  {
    action1();
    action2();
  }

  public void parallelExample()
  {
    Thread otherThread =
      new Thread(new Action1Runnable());
    otherThread.start();
    action2();
    otherThread.join() ;
  )

  class Action1Runnable implements Runnable
  {
    public void run()
    {
      action1();
    }
  }
}
```

Figure 3. Example program that parallelizes two actions using Java threads

The second of the two "other basic solutions" (calls to external library functions) relies on pre-built libraries dedicated to parallelization. One of the most popular standards used for parallel computing is the Message Passing Interface (MPI). Java programs are able to use the MPI with the help of the MPI for Java library. But using the MPI for Java library is somewhat cumbersome (Judd, Clement, Snell, & Getov, 1999). For example, Java's lack of explicit pointers leads to inefficient and confusing code when copying multidimensional arrays and complicated objects. Due to this problem, this study will not compare MPI parallelized programs to aspect-oriented programs that use this study's new loop pointcut.

Another external pre-built library for handling loop parallelization is the Task Parallel Library (TPL), introduced by Microsoft in 2007 (Leijen & Hall) for its ASP.NET framework. The TPL implements a `Parallel.For` method that allows programmers to express their desire to execute a loop in a parallel fashion. For example, Figure 4 shows C# code that uses the `Parallel.For` method to parallelize the calculation of the product of two matrices (Leijen, Schulte, & Burckhardt, 2009).

```
void ParMatrixMult(int size,
  double[,] m1, double[,] m2, double[,] result)
{
  Parallel.For(0, size, delegate(int i)
  {
    for (int j=0; j<size; j++)
    {
      result[i, j] = 0;
      for (int k=0; k<size; k++)
      {
        result[i,j] += m1[i,k] * m2[k,j];
      }
    }
  });
} // end ParMatrixMult
```

Figure 4. Matrix multiplication, using the pre-built `Parallel.For` method

The `Parallel.For` method is particularly efficient in that it abstracts the number of threads used (Leijen et al., 2009). While the number of threads is transparent to the source code, the compiler attempts to match the number of threads to the number of available processors. So in Figure 4, it's not true that each iteration of the `Parallel.For` loop will use its own thread, even though it might appear that way. Such abstraction is beneficial in that it can avoid the use of too many threads and the overhead that comes with them (each thread requires its own stack and bookkeeping information). The TPL's `Parallel.For` method does not attempt to determine whether the designate loop is an appropriate candidate for parallelization. It is up to the programmer to make that determination. One safety measure that is embedded in the `Parallel.For` method is that if the targeted loop throws an exception during the execution of one of the loop's iterations, the loop's remaining iterations are terminated and the thrown exception is passed back to the method that called the failed thread (Leijen & Hall, 2007).

Unfortunately, in parallelizing a program, the explicit parallelization technique and also the calls-to-external-library-functions technique both require significant code refactoring and they both lead to programs that exhibit significant code tangling (Harbulot & Gurd, 2004). Using aspects for parallelization also requires code refactoring (Harbulot, 2006), but as shown in an upcoming subsection, aspects for parallelization can lead to a reduction in code tangling.

Annotations

Several studies have used annotations to address the problem of how to select loop join points for aspectization purposes (Bik, Villacis, & Gannon, 1998; Eaddy & Aho, 2006; Kiczales & Mezini, 2005). For example, Bik, Villacis, and Gannon used commented "par" annotations to specify loops targeted for parallelization. In the following code, the par annotation's threads attribute specifies the number of threads that are to be used in the parallelization:

```
/*par threads=4 */
for (int i=0; i<500; i++)
{ ...
```

There are several benefits to using annotations for aspects. They are relatively easy to understand and implement, and that leads to programs that are more robust and easier to maintain. The associated advice code is more reusable since it can access annotated methods without having to hardcode method details within the advice itself (Kiczales & Mezini, 2005). The primary drawback to using annotations for aspects is that they tend to produce less modular programs, since the cross-cutting concern annotation appears in the core-concern file, not the advice file (Eaddy & Aho, 2006). Since this study's primary goals is to implement a loop pointcut that identifies parallelizable loops, it is

inappropriate to rely on explicit programmer annotations to make such identifications, and this study will refrain from using annotations as part of its solution.

Code Tangling for Parallelized Loops

As noted earlier, traditional loop parallelization techniques (adding thread code directly to the original program code), can create substantial code tangling between the program's original primary concern(s) and the newly introduced parallelization concern (Harbulot & Gurd, 2004). This phenomenon is illustrated by the code shown in Figure 5 and Figure 6. Figure 5, a slightly modified version of (Harbulot & Gurd, 2004, p. 125), shows a `cipherIdea` method, which performs a trivial encryption one byte at a time on a passed-in `text1` parameter. The method uses a `key` parameter to perform the encryption and uses a `text2` parameter to store the resulting encrypted message. Figure 6 shows a refactored version of `cipherIdea`, with embedded parallelization code. Figure 5's `cipherIdea` method implements the encryption concern without parallelization. Figure 6's parallelized `cipherIdea` method also implements the encryption concern, but the encryption concern is surrounded by the parallelization concern. Specifically, Figure 6's encryption concern is limited to the most nested loop inside the `run` method, and the rest of the `cipherIdea` method implements the parallelization concern. Thus, the two concerns are tangled within the `cipherIdea` method.

```
private void cipherIdea(
  byte[] text1, byte[] text2, int[] key
{
  int priorIndex; // index of previously encrypted bit
  for (int i=0; i<text1.length; i+=8)
  {
    // perform encryption on 1 byte of text1
    for (int j=i; j<i+8; j++)
    {
      priorIndex = j>=1 ? j-1: text1.length-1;
      text2[j] = text2[priorIndex] + text1[j] + key[j-i];
    }
  } // end for i
} // end cipherIdea
```

Figure 5. Original implementation of `cipherIdea`, without parallelization

```
private void cipherIdea(byte[] text1, byte[] text2, int[] key)
{
  int max;  // stopping point for text1 index
  int step; // step size between text1 index values
  int priorIndex; // index of previously encrypted bit

  max = text1.length;
  step = NUM_OF_THREADS * 8;
  Thread[] threads = new Thread[NUM_OF_THREADS];

  for (int n=0; n<NUM_OF_THREADS; n++)
  {
    Runnable r = new Runnable()
    {
      public void run()
      {
        for (int i=n; i<max; i+=step)
        {
          // perform encryption on 1 byte of text1
          for (int j=i; j<i+8; j++)
          {
            priorIndex = j>=1 ? j-1: text1.length-1;
            text2[j] = text2[priorIndex] + text1[j] + key[j-i];
          }
        } // end for i
      } // end run
    }; // end anonymous Runnable object
    threads[n] = new Thread(r);
  } // end for

  for (int n=1; n<NUM_OF_THREADS; n++)
  {
    threads[n].start();
  }
  threads[0].run();

  try
  {
    for (int n=1; n<NUM_OF_THREADS; n++)
    {
      threads[n].join();
    }
  } // end try
  catch (InterruptedException e)
  { }
} // end cipherIdea
```

Figure 6. Refactored `cipherIdea`, with parallelization using Java threads

The code tangling problem for parallelized loops can be alleviated with the help of aspect-oriented programming. Figure 7, from (Harbulot & Gurd, 2006, p. 65), shows how

the encryption loop in Figure 5's `cipherIdea` method can be parallelized without code tangling by using around advice from Harbulot & Gurd's LoopsAJ aspect-oriented compiler. Specifically, Figure 7's around advice implements the same parallelization logic as that in Figure 6, but the encryption concern is omitted. With an aspect-oriented system, the encryption concern, found in Figure 5, would be combined with the parallelization concern during the aspect-oriented weaving process. Consequently, with the encryption and parallelization concerns in different source code files, separation of concerns is increased and code tangling is decreased.

```
void around(int min, int max, int step):
  within(LoopsAJTest) &&
  loop() && args(min, max, step)
{
  int numThreads = 4;
  Thread[] threads = new Thread[numThreads];
  for (int i=0; i<numThreads; i++)
  {
    final int t_min = min + i;
    final int t_max = max;
    final int t_step = numThreads * step;
    Runnable r = new Runnable()
    {
      public void run()
      {
        proceed(t_min, t_max, t_step);
      }
    };
    threads[i] = new Thread(r);
  }
  for (int i=1; i<numThreads; i++)
  {
    threads[i].start();
  }
  threads[0].run();
  try
  {
    for (int i=1; i<numThreads; i++)
    {
      threads[i].join();
    }
  }
  catch (InterruptedException e)
  { }
} // end around
```

Figure 7. Loop parallelization advice using LoopsAJ and Java threads

How to Detect Loop Parallelizability

To determine in advance whether a loop is parallelizable, the loop's statements are analyzed for cross-iteration dependence. If cross-iteration dependence is found, then the

loop is not parallelizable. *Cross-iteration dependence* means that the execution of a statement in one iteration of the loop is dependent on the execution of a statement in another iteration of the loop (Blume et al., 1994).

There has been extensive research on dependence analysis in order to enable the parallelization of computer programs. Various solutions have been found for the cross-iteration dependency problem, with varying degrees of complexity and speed (Kyriakopoulos & Psarris, 2004). While the different dependence analysis techniques differ in terms of their computational complexity, they all rely on the need to avoid three types of cross-iteration dependencies – flow dependence, antidependence, and output dependence. Flow dependence is when a statement reads from a memory location that an earlier statement wrote to. Antidependence is when a statement writes to a memory location that an earlier statement read from. Output dependence is when a statement writes to a memory location that an earlier statement wrote to (Blume et al., 1994).

Dependence analysis becomes more difficult in certain circumstances. When loops access arrays, it can be difficult to determine when two array references are dependent on each other. For example, if x[i + 1] and x[j -1] are updated in different loop iterations, dependence analysis requires comparing i + 1 to j – 1 (Blume et al., 1994). Nested loops add to the complexity as well. When array references are used in conjunction with nested loops, the complexity of the dependence analysis problem becomes NP-complete (Kyriakopoulos & Psarris, 2004).

JAVAB

Bik and Gannon developed a prototype parallelization tool named JAVAB that is freely available for educational and research purposes (Bik & Gannon, 1997). They realized that more parallelizable loops could be identified by using runtime analysis, but in the interest of keeping things simple, they relied on compile-time analysis. JAVAB executes either in conjunction with a Java compiler or as a standalone bytecode-to-bytecode transformation tool, to be run after Java compilation.

As part of their keep-it-simple strategy in JAVAB, Bik and Gannon limited the detection of parallelizable loops to just "trivial loops" (Bik & Gannon, 1997). They defined trivial loops precisely using edges and nodes in a control-flow graph. They defined trivial loops more conceptually as a loop with (1) one exit point, (2) a lower bound with a single definition, and (3) an upper bound that is loop-invariant (i.e., the loop's upper bound does not change its value inside the loop body). As described later, this study borrowed from JAVAB's parallelizable loop detection mechanisms and added to them.

Bik and Gannon stated that their JAVAB tool was a prototype, not a robust tool capable of detecting all parallelizable loops. They intended it to be used as a starting point for further parallelizable loop detection development (Bik & Gannon, 1998). In the years since Bik and Gannon's JAVAB tool was introduced in 1997, other researchers have made considerable progress in the area of parallelizable loop detection. The following sections describe some of that progress.

Data Dependency in Scalar-Array-Based Loops

Most cross-iteration dependence analysis studies have focused on detecting parallelizability for loops that handle arrays. The prominence of array-based loops in parallelizable loop studies is due to their penchant for handling numeric calculations, and such numeric calculations tend to benefit the most from parallelization. Bik and Gannon's 1997 JAVAB tool attempted to handle both array-based loops and non-array-based loops, but most loop iteration dependence analysis studies since 2000 have focused exclusively on array-based loops (Aho et al., 2007; Dig et al., 2009; Wu, Feautrier, Padua, & Sura, 2002).There are many real-world programs that process large amounts of numeric data using arrays and are therefore good candidates for loop parallelization. Examples are programs that handle weather forecasting, protein folding for the design of drugs, and fluid dynamics for the design of aeropropulsion systems (Aho et al., 2007).

To determine whether an array-based loop is parallelizable, it is necessary to look for common array accesses that occur at different iterations of the loop. For example, note the code fragment in Figure 8. The first loop's assignment statement, `arr[i] = arr[i] * arr[i];`, accesses the *i*th array element, where *i* is the current loop index. Most studies treat statements that access just the *i*th element and no other elements as being safely parallelizable because different loop iterations operate on different array elements. That is true for scalar arrays (i.e., arrays that hold primitive values), but it's not necessarily true for arrays that hold objects. We'll describe object arrays later on, where we take into account the possibility that distinct array elements hold references to identical memory locations.

```
int[] arr = new int[100];
...
for (int i=0; i<100; i++)
{
  arr[i] = arr[i] * arr[i];
}
for (int i=1; i<100; i+=2)
{
  arr[i] = arr[i-1];
}
```

Figure 8. Code fragment with a parallelizable array-based loop

In Figure 8, the second loop's assignment statement, `arr[i] = arr[i-1];`, accesses both the *i*th and the (*i*-1)th array elements. With multiple array elements accessed, the second loop is more difficult to analyze in terms of cross-iteration data dependence. If the loop index incremented by 1 each time through the loop, then each loop iteration would share an array element access with one other loop iteration. However, since the loop index increments by 2, and the array is scalar (declared with `int[]`), there are no cross-iteration data dependences, and the loop can be safely parallelized.

The data dependence analysis described above is fairly straightforward, but with more complicated array index expressions (more complicated than `arr[i]` and `arr[i-1]`), the analysis can become very difficult. Data dependence analysis problems involving array elements have been studied widely for decades, and such problems are known to be NP-complete (Aho et al., 2007). Determining whether different iterations access the same or different memory locations is made easier if loop limits and array subscript expressions are *affine*. An affine expression is a variable times a constant plus a

constant. For example, the array index in `arr[2*i+3]` is an affine expression, and the array index in `arr[i*j]` is not an affine expression (Aho et al., 2007)

The Banerjee test is a relatively popular test for determining data dependences within a program (Banerjee, Eigenmann, Nicolau, & Padua, 1993; Ricci, 2002). It calculates approximate subscript ranges of elements that can be accessed by each array access instruction. If the ranges overlap and a write access occurs before another access, then a data dependence exists. The ranges are determined by sets of inequalities, which define the bounds of linear/affine expressions.

Data Dependency in Object-Array-Based Loops

Most studies have detected parallelism in arrays by assuming that with $i \neq j$, arr[i] and arr[j] point to two different memory locations. But with arrays of objects, it's important to verify that the two different array elements don't contain (1) references to the same object, or (2) references to two different objects where those objects have fields that point to common objects. In (2009), Dig et al. took those possibilities into account as part of their ReLooper tool, which detects array-based loops that can be safely parallelized and refactors them so they will run in parallel when executed. ReLooper performs the loop parallelization detection on a program's bytecode, whereas it performs the refactorization on the program's source code.

Other loop parallelization studies (Marron, Mendez-Lojo, Hermenegildo, Stefanovic, & Kapur, 2008; Wu et al., 2002) have tracked the entire heap for every object alias, but in the interest of speed, ReLooper just tracks the aliases that appear within the loop arrays targeted for possible parallelization.

In its safety analysis for loop parallelization, ReLooper does not attempt to handle all types of loops. It just handles standard `for` loops and for-each loops, and each loop must iterate over an array or a vector. ReLooper deems a loop to be safely parallelizable if, within the loop, (1) there are no I/O operations, (2) all of the elements in the array (or vector) are traversed, and (3) there are no conflicting memory accesses. To satisfy constraint 2, ReLooper verifies that the loop's index variable traverses up or down by 1 through every element in the array and it disallows the `return`, `break`, and `throw` statements. To satisfy constraint 3, ReLooper verifies that there are no read-then-write, write-then-read, or write-then-write accesses to the same memory location for different iterations of a loop.

In looking for conflicting memory accesses, other studies have taken the approach that method calls are too difficult to analyze, and have consequently assumed that all loops with method calls are non-parallelizable. On the other hand, ReLooper handles method calls by tracing them and their memory accesses as necessary.

ReLooper's analysis of parallelizable loop detection is conservative. That is, it flags all non-parallelizable loops as non-parallelizable (as shown by an empirical study), but it sometimes flags parallelizable loops as non-parallelizable with a warning that they might be parallelizable. With that in mind, users are given the opportunity to override warnings and allow warned-about loops to be parallelized.

Loop Parallelization Speculation

Thread-level speculation (TLS), also called speculative parallelization (SP), is a parallelization technique that attempts to execute loops in parallel even when they are not

flagged by the compiler as being safely parallelizable (Garcia-Yaguez, Llanos, & Gonzalez-Escribano, 2013). They are "speculated" to be parallelizable and during the execution process, the TLS mechanism monitors the operations of each loop iteration executed in parallel. If the TLS mechanism detects any dependencies between the loop iterations, the relevant iterations are stopped and re-executed in the proper order. The re-execution ensures that when a dependency is found, the program still produces its correct result.

In (2008), Zhong, Mehrara, Lieberman, & Mahlke claimed that the current tools for identifying parallelizable loops were inadequate. Specifically, they claimed that tools that focused on finding parallelizable loops in scientific programs (programs with scalar arrays that require dependence analysis across different loop iterations) and also tools that focused on finding parallelizable loops in non-scientific programs (programs with object arrays that require points-to analysis for reference variables) both identified fewer parallelizable loops than could be identified by (intensive) manual analysis. In response to that assessment, Zhong et al. developed a loop parallelization speculation tool named DOALL that identifies loops that have a low probability of memory dependences between different iterations. In running programs through their memory profiler and recording memory accesses, their tool categorizes loops as "speculative DOALL loops" if they contain zero or very few cross-iteration memory dependences, where a memory dependence is defined as two instructions that access the same location in memory. Zhong et al.'s original parallelization tool was able to identify more loops as (speculatively) parallelizable than other tools, but the researchers determined that even more loops could be made parallelizable by introducing compiler transformations which

focused on reducing low-level cross-iteration register and control dependences. They found that such compiler transformations did substantially increase the number of loops that could be parallelized.

In (2009), Hammacher, Streit, Hack, and Zeller conducted a study similar to the Zhong et al. study (2008) in that it also developed a memory profile tool that analyzes program data dependences. In addition to identifying data dependences, the Hammacher tool finds a program's critical path, which is the longest path of instructions in a program that must be executed sequentially. The critical path serves as a baseline lower bound on the execution time, if all other parts of the program are able to execute in a parallel fashion. The Zhong tool processes C programs, while the Hammacher tool analyzes Java programs.

Quite a few studies have implemented various versions of the loop parallelization speculation technique (Hammacher et al., 2009; Sato, Inoguchi, & Nakamura, 2011; Tripp, Yorsh, Field, & Sagiv, 2011; Zhong et al., 2008). Unfortunately, none of those studies provided implementation mechanisms that could be used for this paper's study because they relied on work done after the compilation process. This study implemented a new parallelizable loop pointcut on top of an aspect-oriented compiler. An aspect-oriented compiler does all its work before the end of the compilation process, thus eliminating loop parallelization speculation as a possible technique for use in this study.

Program Dependence Graphs

In (1987), Ferrante, Ottenstein, and Warren introduced a program dependence graph (PDG) as a tool that can be used to help with a variety of program analysis and

refactoring initiatives. In modeling a program's control dependencies, it provides hooks for developers to analyze different parts of a program and possibly optimize those parts. For example, Ferrante, Ottenstein, and Warren showed how their PDG could be used to modify control dependencies relating to loop unrolling operations.

In a recent study (2011), Sadat-Mohtasham relied on PDGs as an underlying mechanism to implement a set of aspect-oriented pointcuts. In compiling a target program with an aspect-oriented compiler, Sadat-Mohtasham's tool creates a PDG, uses the PDG to identify constructs within the target program, and then uses the identified constructs to weave advice into the resulting compiled target program.

In his study, Sadat-Mohtasham implemented loop pointcuts, conditional pointcuts, and a new type of pointcut – a "transactional pointcut" (transcut). A transcut is a general-purpose pointcut that allows a group of pointcuts to be considered as one pointcut – a `transcut` pointcut. A `transcut` pointcut matches a join point if the transcut's contained pointcuts match with join points inside the transcut join point. The contained pointcuts and contained join points must be in the same order, but they do not have to be contiguous. The Results chapter will describe how transcuts were used in the current study. But for now, the focus is on how PDGs are created. An understanding of PDGs was necessary for the current study because the identification of parallelizable loops required using and modifying the PDG-creation code in Sadat-Mohtasham's transcut tool.

Since a PDG implements control dependencies, it's necessary to formally define what a control dependency is. A control dependency is when one program instruction determines whether another program instruction executes. This can be explained best in

the context of a control flow graph (CFG). Note Figure 9, from (Sadat-Mohtasham, 2011, p. 31), which shows an example Java method and its associated CFG. Each of the figure's boxes is a "node," and each node represents a "block" of code. Each block is comprised of one or more "units" that execute in sequential fashion, and each unit is a bytecode instruction. The CFG shows the order of execution of the blocks that form the CFG's method.

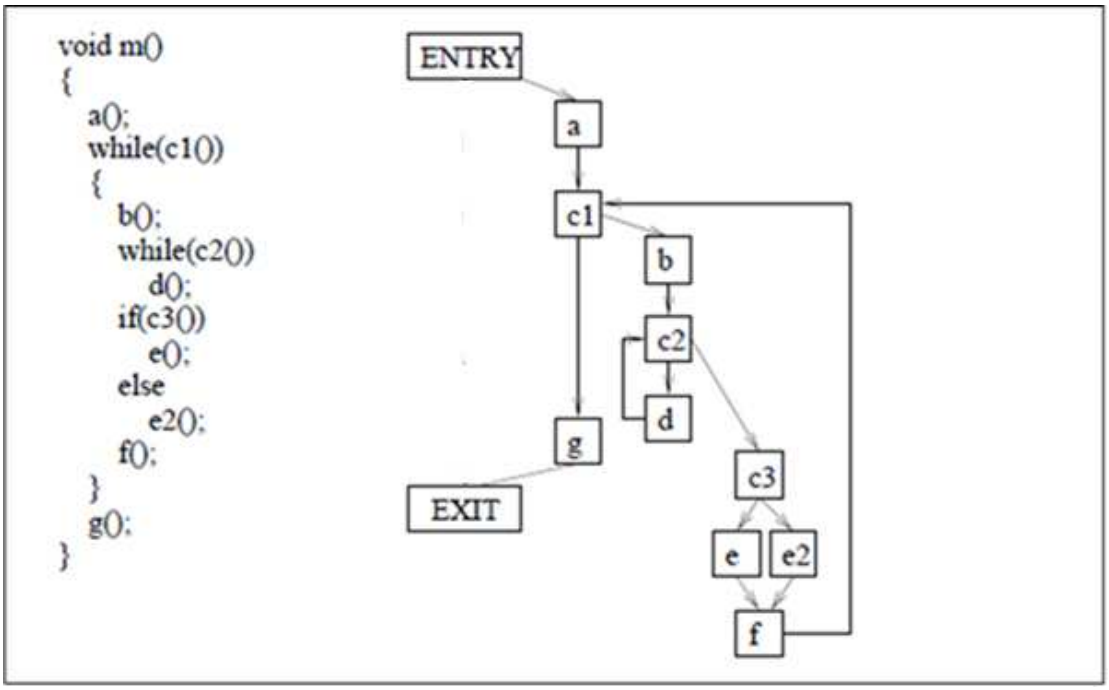

Figure 9. Example method with associated control flow graph

In Figure 9, nodes g and b are control dependent on node c1 because c1's condition value determines which node executes next – g or b. Now let's present the formal definition of control dependency, as stated in (Ferrante et al., 1987, p. 323):

> Let G be a control flow graph. Let X and Y be nodes in G. Y is control dependent on X if and only if:

1) There exists a directed path P from X to Y with any node in P (excluding X and Y) post-dominated by Y, and
2) X is not post-dominated by Y.

The definition of control dependency relies on the definition of one node being "post dominated" by another node. Informally, node W post dominates node V if the only way to get from V to the graph's exit is through W. Here's the formal definition of post dominated, as stated in (Ferrante et al., 1987, p. 323):

> A node V is post-dominated by a node W (not including V) in G if every directed path from V to G's exit node contains W.

Since PDGs rely on control dependencies and control dependencies rely on nodes being post-dominated by other nodes, to create a PDG, it's necessary to first create a post-dominator tree. See the post-dominator tree in Figure 10, from (Sadat-Mohtasham, 2011, p. 32). Its nodes correspond to the nodes shown in Figure 9. In the post-dominator tree, each node is said to post dominate all the nodes that are descendants of it (i.e., post dominance doesn't apply to just the nodes that are immediately below another node).

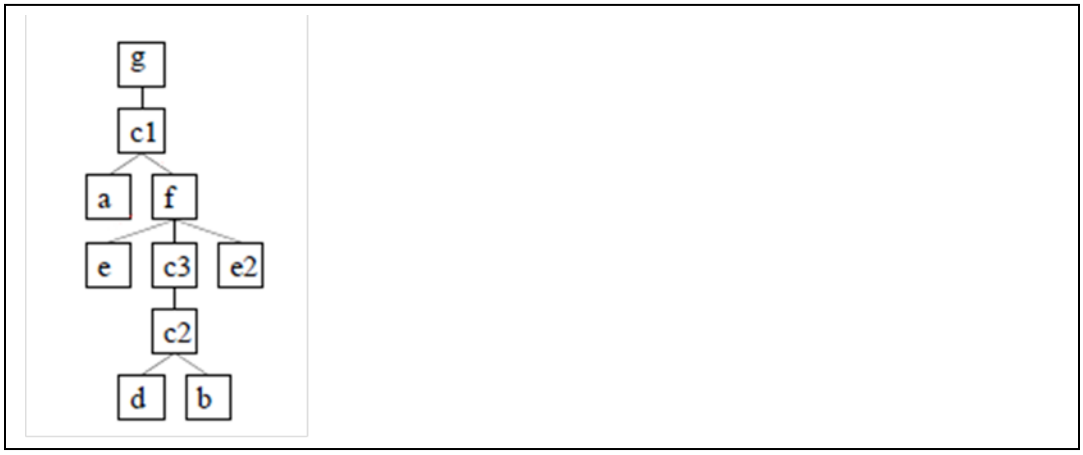

Figure 10. Post-dominator tree for the control flow graph in Figure 9

Figure 11 from (Sadat-Mohtasham, 2011, p. 33) shows the PDG for the method in Figure 9. In the PDG, each arrow represents a control dependency, such that a node at the head end of an arrow is control dependent on the node at the tail end of that arrow. The control dependency arrows come from the definition of control dependency and the method's post-dominator tree.

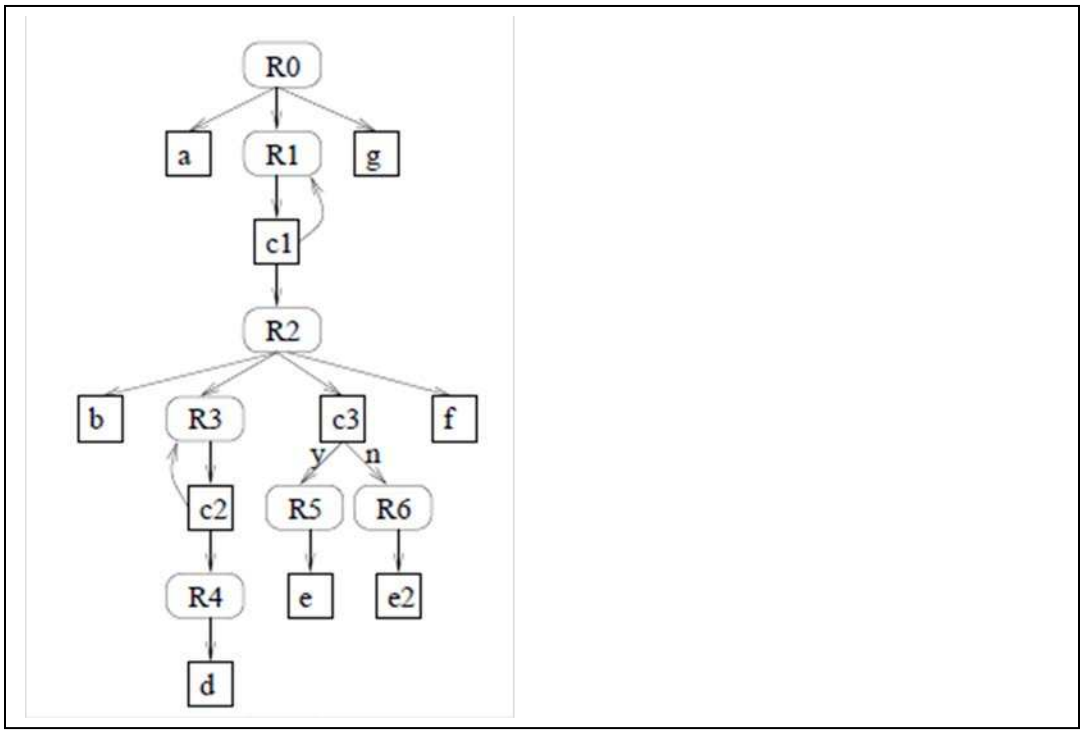

Figure 11. Program dependency graph for the method in Figure 9

There are two types of nodes in a PDG –standard CFG nodes and region nodes (Ferrante et al., 1987). In Figure 11, the region nodes are represented by R0 through R6. In Sadat-Mohtasham's transcut tool, each region node stores data that specifies the CFG nodes that are part of the region node's region. For example, in Figure 11, the R0 region node represents a region containing the a and g CFG nodes.

By definition, each region contains a group of nodes such that during any execution path from the entry point to the exit point, all of the nodes are executed or none of the

nodes are executed. For example, as shown in Figure 9, Region R2's CFG nodes, b, c3, and f, are either all executed (when c1 branches down) or all skipped (when c1 branches up) (Sadat-Mohtasham, 2011).

The purpose of region nodes and regions is to compartmentalize different parts of a program so that the different parts can be analyzed more effectively. For example, this study's parallizable loop pointcut implementation uses a method's PDG regions to identify each loop's starting and ending points and subsequently determine whether the loop has single entry and exit points. The nested nature of PDG regions is particularly helpful for identifying nested loops (Ferrante et al., 1987).

In creating a PDG, CFG nodes are inserted according to their control dependencies. Region nodes are inserted into the CFG so that there is one region node for each branch path from a condition node. For a loop condition node, that means one region node above the condition node (for the loop-back path) and one region node below the condition node (for the loop-termination path). For example, in Figure 11, see nodes R1 and R2 above and below node c1. For an if condition node, both region nodes go below the condition node. For example, in Figure 11, see nodes R5 and R6 below node c3. Note that region nodes are not shared by condition nodes. Thus, in Figure 11, it would be inappropriate to omit node R3 and have nodes c1 and c2 share region node R2. In addition to region nodes being inserted for condition nodes, an R0 region node is inserted at the top of the PDG, to represent the method's starting point.

Abc, Soot, and Jimple

Since this study's purpose was to implement a new pointcut (a parallelizable loop pointcut) for an aspect-oriented compiler, it was important to find an aspect-oriented compiler which was extensible. Because the AspectJ Compiler (ajc) was AspectJ's original compiler, its primary goal was to support AspectJ's specifications correctly. On the other hand, when the aspect bench compiler (abc) was built, its primary goal was to be extensible (Avgustinov et al., 2006). Because of abc's focus on extensibility, the current study used abc as its starting-point aspect-oriented compiler.

The abc compiler was designed and built with the help of two well-established compiler framework tools – polyglot for abc's frontend and Soot for abc's backend. Abc's frontend, implemented with polyglot, provides the interface between abc's aspect syntax and the rest of the compiler. Abc's backend, implemented with Soot, provides the analysis and code generation necessary for the matching and weaving operations inherent in abc's aspect-oriented functionality. The current study relied on abc's polyglot frontend code to define the syntax for the new parallelizable loop pointcuts. The current study relied even more on abc's Soot backend code. In particular, a significant amount of code was added to Soot's backend code in order to enable the matching operations between parallelizable loop pointcuts and parallelizable loop join points. Likewise, code was added to Soot's backend code in order to enable the matching operations between args pointcuts (associated with parallelizable loops) and parallelizable loop join points. Also, code was added to Soot's backend code in order to perform the weaving operations between the parallelizable loops and the multi-threading aspect code. In adding code to

Soot's backend, the study relied heavily on Soot's application programming interface (API). It contains many useful code-manipulation methods.

To make extensions to abc easier to implement, Soot converts target Java programs to instructions from the Jimple instruction set. Jimple's instructions are in between Java source code and Java bytecode in terms of complexity. Compared to the instructions in the Java language, Jimple instructions are simpler and there are fewer of them. That makes it easier for developers to analyze and manipulate a Java program's operations (Einarsson & Nielsen, 2008). For example, in the current study, it was necessary to identify parallelizable loops, regardless of the loop's Java construct – `while`, `do`, or `for`. With Jimple, all three Java loop constructs are represented with the same single construct – an if statement that jumps to the top of the loop.

Compared to the instructions in Java bytecode, Jimple instructions can be easier to understand. Bytecode relies on an implicit stack for its operations, and that can lead to confusion. In particular, by looking at a bytecode instruction, it's hard to know which previous instruction produced the value taken from the stack. And that makes it difficult to create a control flow graph, which is essential for many program analysis studies. On the other hand, Jimple instructions do not rely on an implicit stack for their operations. Instead, they rely on local variables to store data. That makes program analysis and the creation of a control flow graph easier (Lam, Bodden, Lhotak, & Hendren, 2011).

Chapter 3

Methodology

Methodologies Used in Prior Studies

No single prior study was found that matched the complete domain and goals of the current study. However, the studies described in this section shared some of the domain and goals of this study, and, as such, they provided helpful background information.

Bik, Villacis, and Gannon (1998) conducted an experimental study that used annotations to specify loops targeted for parallelization. Using a benchmark set of programs, they compared the performance of their parallelized annotated programs with the performance of standard serial programs. It was similar to the current study in that it attempted to automate the parallelization of loops and it tested using a benchmark set of programs. However, unlike the current study, it did not attempt to detect parallelizable loops on its own, and it did not tie in with an aspect-oriented compiler.

Harbulot and Gurd (2006) conducted an experimental study on a benchmark set of programs where four techniques were used to parallelize the loops in each program. The techniques were compared by evaluating the programs' performances for each of the four techniques. The first three techniques relied on refactoring each loop so that each loop's interface and execution were defined in an external class. By using an external class (RectangleLoopA, RectangleLoopB, and RectangleLoopC for each of the three techniques), the class's instantiation could be tracked by an AspectJ aspect. The RectangleLoopA model relied not only on a RectangleLoopA object, but also on an

object that handled the execution of the loop's body. The RectangleLoopB model attempted to improve performance by putting the loop body's execution code in a subclass of the RectangleLoopB class. That way, external method calls were not required. The RectangleLoopC model attempted to improve performance further by moving nested `for` loop headers into separate methods. That enabled loop unrolling optimizations to occur. The Harbulot and Gurd study's fourth technique relied on a new loop pointcut that they developed for an aspect-oriented compiler. Their loop pointcut meant that programmers did not have to refactor loops into separate classes (e.g., RectangleLoopA, RectangleLoopB, and RectangleLoopC).

Harbulot and Gurd implemented their loop pointcut as part of their new LoopsAJ compiler, which is an extension of the abc aspect-oriented compiler. In implementing LoopsAJ, Harbulot and Gurd chose to use abc rather than the original aspect-oriented compiler, ajc, because abc was specifically designed to be extensible (Harbulot & Gurd, 2006).

Of all the studies that the current study relied on, the study most important to the current study was Sadat-Mohtasham's transcut study. As mentioned earlier, a transcut is a general-purpose pointcut that allows a group of pointcuts to be considered as one pointcut – a transcut pointcut. Like LoopsAJ, the transcut study also relied on abc, because of its penchant for extensibility. The transcut study's stated evaluation strategy was to "[determine whether] transcuts significantly reduce the need for refactoring to expose join points in real existing software" (Sadat-Mohtasham, 2011, p. 73). Sadat-Mohtasham carried out that evaluation strategy by using standard pointcuts as well as transcut pointcuts to try to identify all the statements within existing try blocks without having to

refactor the try block statements. The purpose of identifying such statements is so software developers would be able to use an aspect-oriented compiler to separate the statements that originally came from the `try` block from the error-handling code (i.e., the `try` heading and the associated `catch` block). As hoped for, Sadat-Mohtasham found that transcut pointcuts were better than standard pointcuts at being able to reduce the need for refactoring when attempting to match the statements within the original `try` blocks.

The transcut study's stated evaluation strategy (determining whether transcuts reduce the need for refactoring) does not coincide with the current study's evaluation strategy. However, to implement their stated evaluation strategy, the transcut study was required to determine whether their aspect-oriented transcut pointcuts could be applied to two real-world software systems successfully – with proper pointcut-join point matching and with proper weaving. In that regard, the transcut study's methodology was similar to the current study's methodology.

Overview of the Methodology Used in this Study

This study's goal was to implement a parallelizable loop pointcut as part of an aspect-oriented compiler and show that the parallelizable loop pointcuts and its aspect-oriented compiler were implemented correctly. The study was empirical in nature in that it tested the newly formulated software (the parallelizable-loop-pointcut aspect-oriented compiler) and reported test findings.

The new parallelizable-loop-pointcut aspect-oriented compiler and a standard (non-aspect-oriented) compiler were applied to a subset of the Edinburgh Parallel Computing

Center's (EPCC's) Java Grande Forum Benchmark Suite of programs (Smith & Bull, 2013). The benchmark suite contains coding constructs that occur frequently in scientific software (Harbulot & Gurd, 2004). The benchmark suite contains two sets of "kernels" programs – the programs in the serial set are implemented as sequential programs using standard sequential code, while the programs in the multi-threaded set are implemented as parallel programs using threads. This study compared the multi-threaded programs with refactored versions of those programs. The multi-threaded programs achieved their parallelization via their threads. The refactored programs achieved their parallelization by using an aspect-oriented compiler in conjunction with the study's new parallelizable loop pointcut.

The study determined success based on whether the parallelizable-loop-pointcuts enabled the aspectized programs to run correctly. More specifically, the programs were checked for correctness by (1) comparing the original (non-aspect-oriented) benchmark programs' output to the output from the refactored parallelizable-loop-pointcut programs, and verifying that the outputs were identical. The parallelizable-loop-pointcut compiler was further checked for correctness by (2) verifying that in the refactored parallelizable-loop-pointcut programs, the parallelizable loops were detected and properly parallelized.

The first success constraint was measured by verifying that output was the same from the aspect-oriented and non-aspect-oriented pairs of programs. The Java Grande Forum Benchmark Suite kernel programs implement relatively small (and loop intensive) tasks that would typically be embedded in larger application programs. As such, the kernel programs contain few print statements. Thus, it was necessary to add trace print statements to the programs that allow users to determine whether the programs have

completed their tasks and computed their values correctly. Both the aspect-oriented and non-aspect-oriented programs added the same trace print statements so the aspect-oriented and non-aspect-oriented program pairs should, ideally, generate identical output.

The second success constraint was measured by the parallelizable-loop-pointcut compiler's ability to distinguish between parallelizable and non-parallelizable loops. Each of the tested benchmark programs contained both parallelizable and non-parallelizable loops. The tested benchmark programs were fairly small, and that enabled each of their loops to be manually analyzed for parallelizability. The manual analysis involved examining each loop's control flow and program dependencies. Later in this chapter, the "Detecting Whether a Loop is Parallelizable" section presents an algorithm that describes that manual analysis. For example, only loops with one control-flow exit node are deemed to be candidates for parallelization, so that characteristic was examined manually. As another example, for each use of a local variable within a loop (a variable "use" is when a variable appears in a statement, but not on the left side of an assignment), the loop is deemed to be a candidate for parallelization only if that variable is not assigned a value as part of a subsequent statement within the same loop.

The manual analysis results were compared to the results found by running the parallelizable-loop-pointcut compiler. To verify that the parallelizable loops (and only the parallelizable loops) were found to be parallelizable by the parallelizable-loop-pointcut compiler, and that the parallelizable loops were executed in parallel, trace print statements were added to the loop pointcut aspect code. For each loop that was matched as a parallelizable loop, the print statements indicated the minimum (min) and maximum (max) values for each thread's loop index variable. The min and max values were

interleaved for the different threads, so by reading the printed min and max values, the researcher was able to determine the number of threads executed for a particular loop. The same min and max loop index variable print statements were added to the loop threads in the non-aspect-oriented versions of the benchmark programs. With such print statements in place, finding identical outputs would indicate that (1) parallelizable loops were correctly identified as parallelizable and executed in parallel by the loop-pointcut aspect-oriented compiler, and (2) non-parallelizable loops were correctly identified as non-parallelizable by the loop-pointcut aspect-oriented compiler.

Here are the steps that were used for this study's methodology:

Step 1: Define the syntax for the parallelizable loop pointcut.

Step 2: Implement a structure that enables the detection and identification of loops with one exit point.

Step 3: Implement an algorithm for detecting whether a given loop is safely parallelizable.

Step 4: Implement the weaving of parallelizable loop pointcut aspects into their associated loop join points.

Step 5: Provide pairs of parallelized-loop programs, with each program in a pair having the same functionality, but one is aspect-oriented (using the parallelizable loop pointcut) and one is not aspect-oriented.

Step 6: To determine the correctness of the new parallelizable loop pointcut, compare the output from the parallelizable loop pointcut programs to the output from the non-aspect-oriented programs.

Step 7: To further determine the correctness of the new parallelizable loop pointcut, verify that the aspect-oriented compiler (1) finds the parallelizable loops (and not any non-parallelizable loops) to be parallelizable and (2) executes the loops in parallel.

The first five steps above are rather general in nature. The following sections explain those steps in greater detail.

Defining the Syntax for the Parallelizable Loop Pointcuts

Step 1 involves defining the syntax for parallelizable loop pointcuts. Since this study's loop pointcuts are targeted for parallelizable loops, they are named `loopPar` and `outerLoopPar`. The `loopPar` pointcut is intended to match all parallelizable loops – stand-alone loops, loops that surround other loops, and loops that are nested inside of other loops. The `outerLoopPar` pointcut is intended to match only parallelizable loops that surround other loops. This study implemented an outer loop pointcut and not an inner loop pointcut because parallelizing an outer loop tends to generate greater gains in efficiency (Microsoft, 2012). To justify parallelizing a loop, the overhead necessary to achieve parallelization should be offset by a sufficient amount of work being accomplished in the parallelized loop. An outer loop is more likely to accomplish a sufficient amount of work since an outer loop contains more code than its inner loop.

This study's new parallelizable loop pointcuts are used in conjunction with an `args` pointcut, which identifies a loop's context. Here is the syntax for a `loopPar` pointcut with an associated `args` pointcut:

```
loopPar() && args(min, max, step)
```

All three parameters – `min`, `max`, and `step` – can be accessed within the body of the pointcuts' advice code. The `min` parameter matches the minimum value for the loop's index variable. The `max` parameter matches the maximum value for the loop's index variable. The `step` parameter matches the value that is added to the loop's index variable at the end of each loop iteration.

Detecting Loops with One Exit Node

Step 2 involves detecting loops that have one exit node. As mentioned in the Review of the Literature chapter, Harbulot and Gurd (2006) classified loops into three categories according to their number of exit nodes and number of successor nodes: (1) a loop with one exit node and one successor node, (2) a loop with more than one exit node and one successor node, and (3) a loop with more than one successor node. They determined that only loops in the first category, one exit node and one successor node, were capable of being matched with a loop pointcut that supported (1) before, after, and around advice, and (2) context exposure. (Context exposure means that args pointcuts can be used in conjunction with loop pointcuts.) The current study has the same goal of implementing a loop pointcut that supports before, after, and around advice, and context exposure. Thus, in this study, finding loops that fit into the category of one exit node and one successor node was necessary. To simplify that process, it's important to realize that with Java (source code, bytecode, and Jimple), if there is one exit node, then there must be only one successor node. That's because Java conditional statements are limited to one branching target. So an exit node (implemented with a conditional statement) either goes to a statement within the loop or goes to a statement outside of the loop (to the single

successor node). Thus, in trying to identify loops that have a single exit node and a single successor node, this study was able to limit the identification process to finding loops with a single exit node.

Correctly identifying loops with one exit node was necessary to enable the matching process between the parallelizable loops in a given program and the parallelizable loop pointcuts that might exist in the program's aspect file. Such matching is an example of *shadow matching*, which is what an aspect compiler does to identify join points. To identify loops, the compiler creates a control-flow graph from its bytecode (Harbulot & Gurd, 2006). Note the example control-flow graph in Figure 12.

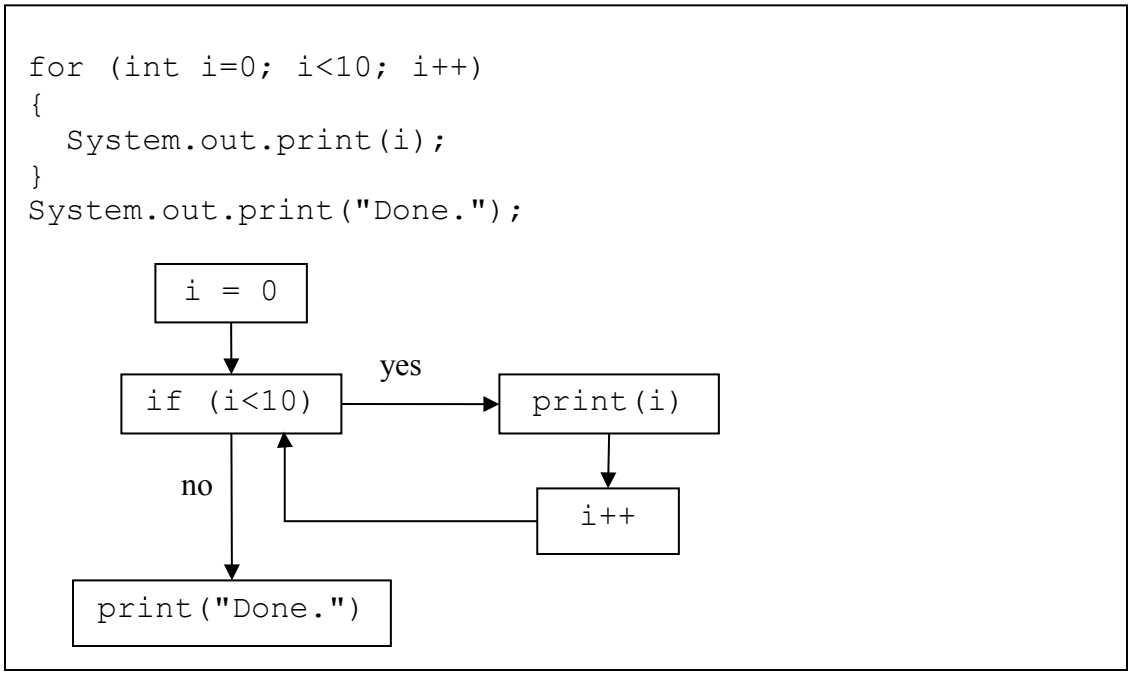

Figure 12. A 1-exit-node, 1-successor-node loop with its associated control-flow graph

After creating the control-flow graph, this study's aspect-oriented compiler looks for dominators, back edges, and natural loops within the control-flow graph. A node x is a *dominator* of node y (i.e., it *dominates* node y) if for every path from the starting node to node y, the path goes through node x. An edge from y to x is a *back edge* if x dominates

y. A *natural loop* for back edge y to x is the set of nodes along a path from x to y, excluding the paths that revisit node x (Aho et al., 2007; Harbulot & Gurd, 2006). After identifying the natural loops, the compiler then determines whether each loop is of the one-exit-node variety. This determination process requires knowing the number of exit nodes for each loop. The compiler checks for the number of exit nodes by examining the program's control-flow graph. Specifically, for exit nodes, the compiler counts each node that has an arrow going from it to outside of the loop.

Detecting Whether a Loop is Parallelizable

Step 3 determines whether a particular loop is parallelizable. The following list provides an overview of the constraints checked for in this study's loop parallelization detection algorithm:

1. The loop must be trivial.
2. Limited uses and definitions for local scalar variables.
3. No field modifications.
4. Safe method calls only.
5. No array dependencies between different elements of an array.
6. For each array reference definition, the array reference's index expression must include the loop's index variable.

The constraints numbered 1, 2, 3, and 5 above came from Bik and Gannon's loop parallelization detection algorithm (1997). The algorithm was presented as part of the documentation for their javab bytecode parallelization tool. Bik and Gannon stated that

their algorithm takes a conservative approach. If their algorithm indicates that a loop is parallelizable, then the loop is assured to be parallelizable. But for some parallelizable loops, their algorithm is unable to identify them as being parallelizable. This limitation is due to the difficulty of solving the parallelizable-loop-detection problem completely. By using their conservative approach, it is easier to verify that certain loops (the ones that follow the constraints above) are parallelizable.

An example of a parallelizable loop that would not be identified as parallelizable by Bik and Gannon's algorithm (and this study's algorithm as well) is a loop that follows all of the constraints shown above except for one part of the first constraint. The first constraint requires the loop to be "trivial." As explained later, one of the requirements for a "trivial loop" is that its index variable is assigned just one time within the loop (e.g., i = i + 1;). Suppose the loop index variable is assigned to itself (e.g., i = i;). In that case, the loop parallelization detection algorithm would not identify the loop as parallelizable even though the loop would actually be parallelizable.

Since Bik and Gannon's javab tool is not part of an aspect-oriented compiler and it does not use Jimple, javab's implementation of the constraints above differs from this study's implementation. The Results chapter provides a finer-grained explanation of this study's loop parallelization detection algorithm implementation details.

Weaving Parallelizable Loop Pointcut Aspects into Loop Join Points

Step 4 in this study's methodology weaves parallelizable loop pointcut aspects into parallelizable loop join points. To implement the weaving process, this study relies on the weaving functionality built into the abc aspect-oriented compiler. The abc compiler uses

the Soot software framework tool to generate Jimple code from Java source code (Avgustinov et al., 2006). Abc then performs the weaving process on the resulting Jimple code. Thus, as with the parallelizable loop matching process, the weaving process requires analyzing Jimple code. In addition, the weaving process requires adding Jimple code. After the weaving takes place, abc converts the resulting Jimple code to bytecode.

Jimple is a typed, 3-address language with no reliance on a stack. With such characteristics, each Jimple instruction explicitly shows the instruction's operation. Such explicit instructions make Jimple easier to work with during the matching process and the weaving process. On the other hand, if bytecode were used for these processes, there would be a need to determine the stack's contents for every instruction that relied on the stack. Also, during the weaving process, there would be a need to add bytecode to have the stack behave appropriately with the newly woven code. For aspect code that uses arguments (i.e., the `args` pointcut), using Jimple provides the benefit of having the arguments readily available as Jimple parameters during the weaving process. On the other hand, if bytecode were used, then retrieving arguments would be more difficult – it would require searching through the stack to find them (Avgustinov et al., 2006).

Creating Programs with Parallelized Loops

Step 5 in this study's methodology creates parallelized loops for testing purposes. To test the effectiveness of this study's proposed parallelizable loop pointcut, the study compared aspect-oriented parallelized loops (using the new parallelizable loop pointcut) versus non-aspect-oriented parallelized loops. This study borrowed from Harbulot and Gurd's strategy and utilized the *worker object creation pattern* as the basis of its loop

parallelization implementations (2006). The worker object creation pattern was first described by Laddad in (2003) as the replacement of a method call with code that (1) instantiates an object that contains the method as one of the object's members, and (2) calls the object's method. Laddad claimed the purpose/benefit of the worker object creation pattern was to be able to treat the method call as an object. Treating it as an object means that the method can be passed to other methods, stored, and called. Another benefit (and a more important benefit for this study) is that the pattern works well with threads. Threads were used in all the worker object creation pattern examples found as part of this study's literature review (Harbulot & Gurd, 2006; Laddad, 2003). All of the examples used a thread object to store the method. The benefit to using a thread is that the called method can be executed in parallel with other threads. Typically, multiple method calls would be encapsulated in multiple thread objects, so that method calls can run in parallel.

The code required to implement the worker object creation pattern is non-trivial, and, as such, if the pattern were used repeatedly, there would be a significant amount of code redundancy. In (2006), to avoid the code redundancy, Harbulot and Gurd used an aspect to implement the pattern. Specifically, they aspectized loops by using around advice for their targeted loops. They executed each loop iteration by embedding a `proceed` method call inside the thread object. Each `proceed` method call executed a subset of the loop's iterations in a separate thread. This study used that same methodology when creating aspectized loops for the comparison of aspectized parallelized loops versus non-aspect-oriented parallelized loops.

For an example implementation of that methodology, see Figure 13 and Figure 14. Figure 13, from (Harbulot & Gurd, 2006, p. 65), shows three loops that are not parallelized. Figure 14, identical to Figure 7 except that Harbulot and Gurd's loop pointcut is replaced with this study's `loopPar` pointcut, shows an aspect file that is intended to parallelize Figure 13's loops without code tangling The three loops in Figure 13 have the same loop signature with three `int` values for min, max, and step, and that signature matches Figure 14's `args` pointcut with three `int` variables for `min`, `max`, and `step`. Therefore, Figure 14 applies to all three of Figure 13's loops. Note how Figure 14's advice uses four threads to execute each loop. The four threads use `t_min` to interleave each thread's loop iteration subsets. For this study's multiple-thread tests, the researcher decided to use four threads because of the current popularity of quad-core CPUs (e.g., AMD Phenom II X4). Testing with four threads should provide the same insights as testing with more than four threads.

```
int MAX = 100 ;
for (int i=0; i<MAX; i++)
{
  /* A */
}

int j = 0 ;
int STRIDE = 1 ;
for (; j<MAX; j+=STRIDE)
{
  /* A */
}

int k = 0 ;
while (k < MAX)
{
  /* A */
  k++;
}
```

Figure 13. Three equivalent loops that are parallelized by **Figure 14**'s advice

```
void around(int min, int max, int step):
  within(LoopsAJTest) &&
  loopPar() && args(min, max, step)
{
  int numThreads = 4;
  Thread[] threads = new Thread[numThreads];
  for (int i=0; i<numThreads; i++)
  {
    final int t_min = min + i;
    final int t_max = max;
    final int t_step = numThreads * step;
    Runnable r = new Runnable()
    {
      public void run()
      {
        proceed(t_min, t_max, t_step);
      }
    };
    threads[i] = new Thread(r);
  }
  for (int i=1; i<numThreads; i++)
  {
    threads[i].start();
  }
  threads[0].run();
  try
  {
    for (int i=1; i<numThreads; i++)
    {
      threads[i].join();
    }
  }
  catch (InterruptedException e)
  { }
}
```

Figure 14. Loop parallelization advice using the proposed `loopPar` pointcut

Resource Requirements

Five primary resources were needed to implement and test this study's proposed parallelizable-loop-pointcut compiler: source code for the transcut software tool, source

code for the abc compiler, the AspectJ compiler, the Java Grande benchmark suite of test programs, and a computer that handles multi-threaded Java programs.

To test this study's new parallelizable-loop-pointcut compiler on a standard set of programs, this study used the EPCC group's Java Grande Forum benchmark suite of programs. Specifically, this study used four programs from the "kernels" set of Java Grande Forum benchmark programs – seriesTest, cryptIdeaTest, sor, and sparseMat.

To test this study's parallelizable loop pointcuts, a single computer that handles multi-threaded programs was sufficient. With such an environment, parallelized loops were able to run subsets of their iterations on separate threads. Using an actual multiprocessor computing environment could have tested execution speed more effectively, but testing for execution speed fell outside of the scope of this project.

Summary

This Methodology chapter described how the investigation was conducted. It first presented methodologies used in several related studies. It then provided an overview of the steps necessary to carry out this study's methodology. It fleshed out details of the more involved steps, such as defining the parallelizable loop pointcut's interface and functionality, detecting loops with one exit node and one successor node, detecting whether a loop is parallelizable, and creating programs with parallelized loops. The Methodology chapter then presented the formats that were used in presenting the results. Finally, the chapter presented the study's resource requirements.

Chapter 4

Results

Introduction

This study's primary goal was to implement a parallelizable loop pointcut, The study's primary question was whether the parallelizable loop pointcut could be implemented correctly. This chapter provides results that show that such a loop pointcut can be implemented correctly.

Detecting whether a loop is parallelizable is one of the key steps in implementing a parallelizable loop pointcut for an aspect-oriented compiler. The design and implementation of a parallelizable loop detection algorithm was one of the significant outcomes of this study. As such, this chapter provides an in-depth explanation of the parallelizable loop detection algorithm.

Another outcome of this study was the determination of how transcut pointcuts and parallelizable loop pointcuts are used within an aspect file in order to check for parallelizable loops and weave the file's multi-threading advice code into the core concerns of target Java programs. This chapter provides an example aspect which illustrates that pointcut usage.

Since the detection of loop parallelizability is known to be an NP-complete problem (Aho et al., 2007), this study made no attempt to implement a pointcut that would detect every loop that was parallelizable. With that in mind, there was no attempt to test the new parallelizable loop pointcut for applicability with an exhaustive set of loops. Instead, the

new parallelizable loop pointcut was tested on four benchmark programs. Specifically, the study used the parallelizable loop pointcut to apply multi-threading advice to a benchmark set of programs that were known to be parallelizable. There were two versions of each benchmark program – (1) an aspect-oriented version, where the aspect-oriented compiler's weaver added the multi-threading functionality, and (2) a non-aspect-oriented version, where the benchmark program's source code directly implemented the multi-threading functionality.

After describing this study's parallelizable loop detection algorithm and the manner in which transcut and parallelizable pointcuts are used, this chapter describes the general purpose of each of the four benchmark programs. It then shows that for each benchmark pair of programs, the aspect-oriented version produced the same or nearly the same results as the non-aspect-oriented version. Next, the chapter provides the results that show that the new parallelizable loop pointcut was able to identify parallelizable loops in two of the four benchmark programs and weave multi-threading advice into the loops identified as parallelizable. For the other two benchmark programs, the parallelizable loop pointcut was unable to identify parallelizable loops. The chapter explains why the parallelizable loop pointcut was unable to identify those parallelizable loops as parallelizable. The chapter then discusses several unexpected problems that arose due to problems inherent in the preexisting software that this study's solution relies on. Finally, the chapter provides a description of the repository that holds the software used to build the study's parallelizable loop pointcut aspect-oriented compiler.

Detecting Whether a Loop is Parallelizable

The following list was first presented in the Methodology chapter. It provides an overview of the constraints checked for in this study's loop parallelization detection algorithm:

1. The loop must be trivial.
2. Limited uses and definitions for local scalar variables.
3. No field modifications.
4. Safe method calls only.
5. No array dependencies between different elements of an array.
6. For each array reference definition, the array reference's index expression must include the loop's index variable.

The rest of this subsection describes each of the six constraints in detail.

Constraint 1. The loop must be trivial.

For the loop to be trivial, these characteristics must hold:

a) One exit node.
b) The loop condition's index variable must not be assigned anywhere within the loop's body other than right before the branch at the bottom of the loop.
c) The loop condition compares its index to an integer constant or to a variable that's not assigned anywhere within the loop.
d) The instruction that occurs right before the branch at the bottom of the loop must be an instruction that increments the index variable by a positive integer constant.

Constraint 1a requires loops to have one exit node. As explained in the previous chapter, that is necessary for implementing a loop pointcut that supports before, after, and around advice, and context exposure. Constraints 1b, 1c, and 1d require that the loop index variable increments in a simple, consistent manner. Such simple, consistent incrementation makes it easier to parallelize such loops by assigning subsets of the loops' iterations to different threads (Harbulot & Gurd, 2004).

Constraint 2. Limited uses and definitions for local scalar variables.

A "local scalar variable" is a primitive variable declared within a method. A "use" of a variable is an instruction that reads the value of a variable. A "definition" of a variable is an instruction that assigns a value to a variable.

Local scalar variables must have limited uses and definitions as follows:

a) For each use of a local scalar variable (but not including the loop's index variable) within the loop, each of the variable's definitions must satisfy one of the following:

   i. The definition is in the same block as the usage block, and before the usage instruction.

   or

   ii. The definition is in a different block as the usage block, and within the same loop, and $v' \in Dom(v)$, where v is the usage block, v' is the block that contains the definition, and Dom(v) is the set of blocks that dominates block v.

   or

iii. The definition is outside the loop.

and

b) All scalar variables defined in the loop must be dead upon exiting the loop.

Constraint 2a requires that for every use of a local scalar variable, the variable gets assigned its value either (1) within the loop and before the variable's use, or (2) outside of the loop.

Constraint 2b requires that for each local scalar variable assigned within the loop, for each use of that definition, the use must appear within the loop, not after the loop. If the use appeared after the loop, then the value for the variable's use would be dependent on which loop iteration executed last. If the loop were executed in parallel, the determination of which loop iteration executed last would be unpredictable, so the use's value would be unpredictable. With such an unpredictable result, the parallel-version loop would not be functionally equivalent to the parallel-version loop. Consequently, the loop should not have been deemed parallelizable.

This constraint assumes that the implementing language uses separate copies of local scalar variables within each thread. That is indeed the case for Java (Jenkov, n.d.). With that in mind, it is acceptable to have a definition of a variable before a use for that variable because then there's no risk of different threads using the other thread's definition. But on the other hand, if a variable use appears before its definition, then it's possible that one thread finishes before another thread starts and then the late-starting thread will use the finished thread's definition, which might cause logic errors.

Constraint 3. No field modifications.

A field can be an instance variable or a class variable. If field modifications were allowed within candidate parallelizable loops, then points-to analysis would be required to find the set of uses for a field definition. Such points-to analysis can add significantly to the analysis's complexity (Wu et al., 2002). Points-to analysis is when static code is traced to determine the object (or set of objects) that a particular reference will point to (or might point to) during the program's execution.

If a field were defined (assigned a value) within a loop, being able to track down all the uses for that field later on is no guarantee that there is a dependence between the uses and the definition. That's because after the original assignment into the field, the field's reference might be assigned a different object. If that were the case, then later uses of the field would access the field from a different object, not from the object whose field was assigned originally.

Constraint 4. Safe method calls only.

Bik and Gannon's parallelizable loop detection algorithm used a constraint of no method calls at all inside the target loop (1997). Their reasoning was that method calls make it difficult to find the set of uses for a particular definition within the loop. That difficulty is exacerbated for called methods that contain field modifications and/or their own method calls. As explained above, if there were field modifications within the called method, tracking down the uses for such field modifications would require points-to analysis, which is known to be a relatively complex process. If there were method calls within the called method, then the analysis would need to take into account the entire

chain of potential method calls, including recursive method calls. That could add to the analysis's complexity significantly. By not allowing method calls, Bik and Gannon's analysis process was simplified significantly.

In testing this study's algorithm on benchmark programs, it became apparent that having a constraint of no method calls at all inside the target loop caused many of the targeted benchmark programs' loops to fail the parallelizable loop test even though the loops could be manually identified as parallelizable. To reduce the number of false negatives, the researcher modified the constraint such that it allows "safe" method calls within the target loop, but not "unsafe" method calls. A method call is considered "unsafe" if (1) the called method contains a field modification(s), (2) the called method contains a method call, or (3) the method call contains an argument(s) that is a reference type. The first two criteria for unsafeness are constraints on their own and have been explained previously. The third criterion (having an argument that is a reference type) is considered unsafe because the reference means that (1) the algorithm would be required to track down possible field modifications to the reference argument's object and (2) the reference could be an array, in which case the algorithm would be required to track down array element definitions and uses within the called method in order to verify compliance with constraints 5 and 6.

This study's parallelizable loop detection algorithm considers standard Java library methods to be safe. This is a reasonable assumption since standard Java library methods know nothing about variables in the loops and programs that are being tested. Thus, if a standard Java library method call is encountered, the algorithm makes no attempt to look for (1) field modifications within the called method, (2) called methods within the called

method, or (3) reference arguments being passed to the called method. Recognizing standard Java library methods as safe is particularly useful for testing purposes, where print methods are normally embedded in loops. Recognizing standard Java library methods as safe is also useful for number-crunching programs, where `Math` class methods are often embedded in loops.

Constraint 5. No array dependencies between different elements of an array.

Looking at all the array accesses that use the loop's index variable i for the array element's index, if there is at least one such access that is a definition (that's when the access is at the left side of an assignment), then determine the lower bound index offset (l for arr[i + l]) and the upper bound index offset (u for arr[i + u]) for all the accesses for a particular array. Then, for the array to be parallelizable, this must hold:

$u - l < step$,

where step is the amount added to the loop's index variable each time through the loop.

When calculating each u and l pair, use all array references that refer to the same array, taking aliases into account. This requires comparing each loop array reference to all the other loop array references, while checking for array bases being aliases of each other and checking for array index expression variables being aliases of each other.

Note the code fragment in Figure 15. It shows a loop that has two aliases, `arr1` and `arr2`, for the same array. The array has two references, `arr1[i]` and `arr2[i-1]`, with an upper bound of 0 and a lower bound of -1, respectively. The loop increments by 1 (`i++`), so step equals 1. Thus, $u - l < step$ does not hold, and the loop is not parallelizable. Below the loop code fragment, Figure 15 shows what would happen if the

loop was (incorrectly) deemed to be parallelizable, and the loop's iterations were executed on separate threads. The left picture shows that if the iterations execute in the order i = 0, i = 1, and then i = 2, the array ends up with values of 1, 2, and 2 in its three elements. On the other hand, if the iterations execute in the order i = 2, i = 1, and then i = 0, the array ends up with values of 0, 1, and 2 in its three elements. The different final values in the array indicate that the loop should be deemed non-parallelizable.

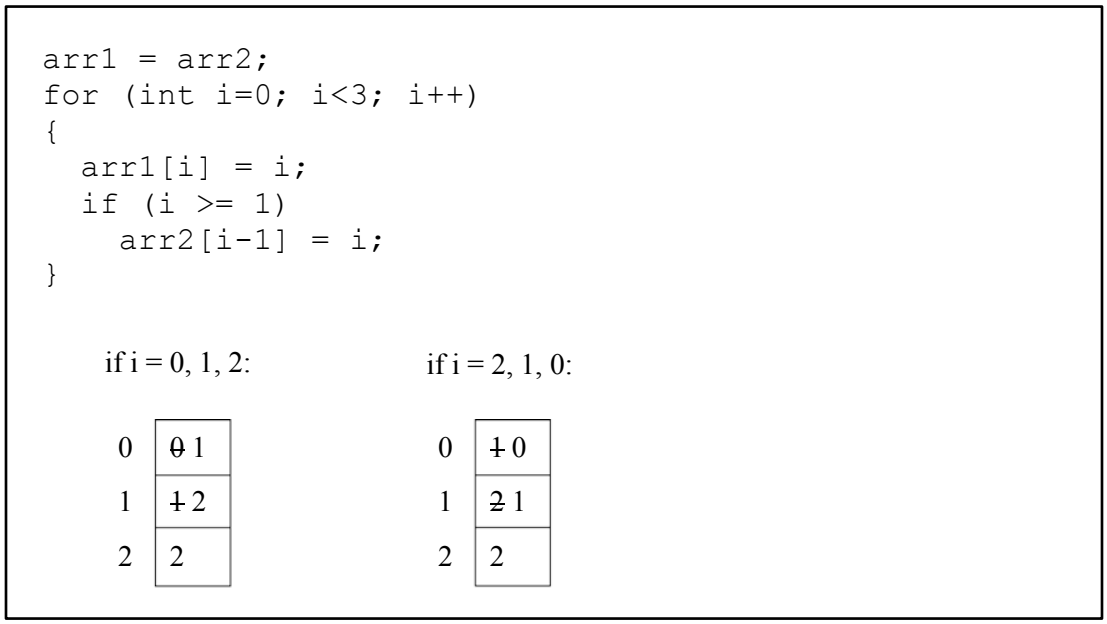

Figure 15. A loop with array references where u – l ≮ step does not hold

If two array references' array bases are method parameters or aliases to method parameters, and the array references' array bases are not aliases of each other, and the array references' offsets are such that u – l ≥ step, then warn the user: If the relevant method parameters are aliases of each other, then the loop's iterations should not be run in parallel.

If an array reference uses an index expression that does not fit the format <for-loop-index> + <integer-constant> or <for-loop-index> - <integer-constant>, then the array

reference cannot be analyzed in terms of its upper or lower bound. In that case, the array reference's containing loop is deemed non-parallelizable.

Constraint 6. For each array reference definition, the array reference's index expression must include the loop's index variable.

By having the loop's index variable in the array reference's index expression and by following Constraint 5 above (no array dependencies between different elements of an array), it ensures that a different array element is updated for each loop iteration. Having a different array element being updated is necessary because if the same array element were updated in different loop iterations, then after the loop finished executing in parallel, it would be unpredictable as to which value was assigned into that array element. For example, assume i is the index variable for a loop, and this statement appears within the loop:

```
arr[3] = i;
```

If the `arr[3]` element were used after the loop, then the use's value would be unpredictable. That behavior is different from the behavior of the loop if it were executed sequentially. Thus, the loop should have been deemed non-parallelizable.

If there were no uses of the array reference in the loop or after the loop, then parallelization would be allowed, but (1) it's difficult to search for uses of an array reference (because of the potentially complex array index expression), and (2) there is no purpose to having a definition for an array reference and not having any uses for it. Thus, this constraint does not require a search for a use for such an array reference.

Transcut Pointcut and Parallelizable Loop Pointcut Usage

The first thing that the parallelizable loop detection algorithm does is identify groups of statements that form loops. The implementation for that identification process comes from the transcut software tool. As explained in the Methodology chapter, the transcut tool's `transcut` pointcut groups other pointcuts together so that a group of join points can be matched by the `transcut` pointcut. In implementing the transcut tool, Sadat-Mohtasham implemented not only the `transcut` pointcut, but also the `looped` pointcut, which can be placed inside of a `transcut` pointcut to match loops (Sadat-Mohtasham, 2011). This study implemented the `loopPar` and `outerLoopPar` pointcuts by extending the transcut tool's `Pointcut` class. The `looped` pointcut is also derived from the `Pointcut` class. When used in aspect files, all pointcuts formed from the subclasses of the `Pointcut` class must be enclosed in a `transcut` pointcut, and that is indeed the case for the new `loopPar` and `outerLoopPar` pointcuts.

Figure 16 shows the aspect used to parallelize loops using the study's parallelizable loop pointcut. Most of the code shown is identical to the multi-threading aspect code shown in the Review of the Literature chapter's Figure 7, but the transcut code at the top of Figure 16 is new. Note that the transcut construct encloses only one pointcut – the `loopPar` pointcut. Transcuts are normally used to enclose a group of pointcuts, but for the purposes of this study, they enclose only one pointcut – a `loopPar` pointcut or an `outerLoopPar` pointcut.

```
public aspect LoopParTestAspect
{
  transcut tc()
  {
    // pointcut loop: outerLoopPar();
    pointcut loop: loopPar();
  }

  void around(int min, int max, int step):
    tc() && args(min, max, step)
  {
    int numThreads = 4;
    Thread[] threads = new Thread[numThreads];

    for (int i=0; i<numThreads; i++)
    {
      final int t_min = min + i;
      final int t_max = max;
      final int t_step = numThreads * step;

      Runnable r = new Runnable()
      {
        public void run()
        {
          proceed(t_min, t_max, t_step);
        }
      };
      threads[i] = new Thread(r);
    }

    for (int i=0; i<numThreads; i++)
    {
      threads[i].start();
    }

    try
    {
      for (int i=0; i<numThreads; i++)
      {
        threads[i].join();
      }
    }
    catch (InterruptedException e)
    { }
  } // end around
} // end aspect LoopParTestAspect
```

Figure 16. Advice that uses a `transcut` pointcut and a `loopPar` pointcut to replace parallelizable loops with multi-threaded versions of those loops

In Figure 16, the transcut construct at the top defines a `loopPar` pointcut with the name `tc` that matches all parallelizable loops. The commented-out `outerLoopPar`

would match all parallelizable loops that were not nested inside other loops. In Figure 16's `around` advice, the `tc` pointcut is `&&`'d with an `args` pointcut. Since the `tc` pointcut represents a `loopPar` pointcut, the `args` pointcut retrieves the `min`, `max`, and `step` values for a matched parallelizable loop of this form:

```
for (int i=min; i<max; i+=step)
```

The `around` advice's code generates four threads and distributes the original loop's iterations between each of the threads. To interleave the iterations between the threads, the advice code uses the `min`, `max`, and `step` values to create `t_min`, `t_max`, and `t_step` values for the loops in each of the generated threads. The first thread uses a `t_min` value of 0, the second thread uses a `t_min` value of 1, etc. All the threads use a `t_step` value of 4. Thus, the first thread uses loop index values of 0, 4, 8, 12, etc. And the second thread uses loop index values of 1, 5, 9, 13, etc.

Description of Benchmark Programs with Parallelizable Loops

This study tested its new parallelizable loop pointcut on four benchmark programs from the Java Grande Forum Benchmark Suite (Smith & Bull, 2013). The four tested programs came from the "Kernels" subset of the benchmark suite of programs. The Kernels programs were chosen because they are fairly short and their computationally intensive algorithms lent themselves well to multi-threaded loop implementations. For the purposes of this study, the four tested programs don't have to be understood fully, but they do need to be understood at a high level, in order to make the results data more comprehensible. What follows are high-level descriptions of the four tested programs.

The “Series” kernel program calculates the Fourier coefficients of the equation $f(x) = (x+1)^x$ over the interval $0 \leq x \leq 2$. The first n Fourier coefficients are calculated by a loop that iterates n times. Each loop iteration executes independently of the other loop iterations, so that loop is parallelizable, and that loop is the one targeted for parallelization by the study’s parallelizable loop pointcut.

The “Crypt” kernel program performs encryption and decryption using the IDEA (International Data Encryption Algorithm) algorithm. The algorithm uses several loops that are good candidates for parallelization. Each of the loops iterates n times through one or more arrays, where each array contains n elements. For example, the algorithm encrypts each of n byte-length messages, with each byte stored as an element in an n-element array.

The “SOR” kernel program performs the successive over-relaxation (SOR) algorithm in solving a linear system of equations. The SOR algorithm loops through a two-dimensional array and updates array elements by relying on neighboring elements’ values. Since the neighboring elements’ values might have been updated in an earlier loop iteration, the algorithm is inherently sequential in nature, not parallel. However, parallelism is made possible by refactoring the algorithm so that there’s synchronization between elements and their neighbors.

The “SparseMatMult” kernel program performs matrix multiplication. It finds the product of all of the non-zero elements in an n x n sparse array. The SparseMatMult algorithm loops through all the rows in the array as part of the multiplication operation. To parallelize the algorithm, groups of the loop’s iterations are split among different threads. To help with efficiency, the number of rows processed by each thread should be

about the same. This partitioning of the rows to different threads is made more difficult by the fact that the matrix is sparse, and only the rows with non-zero elements should be processed. The partitioning of the rows is necessary for parallelization, but such partitioning can possibly cause multiple threads to update the same element in the resulting multiplication matrix. Such redundant updates can introduce small errors into the calculation.

Verification that Aspect-Oriented Benchmark Programs Generate Correct Results

For each benchmark pair of multi-threaded programs, to determine whether its aspect-oriented version produced the same results as its non-aspect-oriented version, the two program versions' outputs were compared. Figure 17 shows those output comparisons. Note that for each pair of programs, the results were the same or very similar for the aspect-oriented and non-aspect-oriented versions of the programs.

	Abbreviated trace output for aspect-oriented program	Trace output for non-aspect-oriented program
Series	coefficient 0,0 Computed value = 2.8729524964837996 Reference value = 2.8729524964837996 coefficient 1,0 Computed value = 0.0 Reference value = 0.0 coefficient 0,1 Computed value = 1.1161046676147888 Reference value = 1.1161046676147888 coefficient 1,1 Computed value = -1.8819691893398025 Reference value = -1.8819691893398025 coefficient 0,2 Computed value = 0.34429060398168704 Reference value = 0.34429060398168704 coefficient 1,2 Computed value = -1.1645642623320958 Reference value = -1.1645642623320958 coefficient 0,3 Computed value = 0.15238898702519288 Reference value = 0.15238898702519288 coefficient 1,3 Computed value = -0.8143461113044298 Reference value = -0.8143461113044298	same same same same same same same same
Crypt	1st original byte: 1 1st decrypted byte: 1 1st original byte: 0 1st decrypted byte: 0 1st original byte: 2 1st decrypted byte: 2 1st original byte: 3 1st decrypted byte: 3	1st original byte: 0 1st decrypted byte: 0 1st original byte: 2 1st decrypted byte: 2 1st original byte: 1 1st decrypted byte: 1 1st original byte: 3 1st decrypted byte: 3
SOR	Validation succeeded Gtotal = 0.4984199298207158, deviation = 0.0	Validation succeeded Gtotal = 0.4984222429146549, deviation = 2.313093939110278E-6
SparseMatMult	Validation succeeded ytotal = 75.02484945753453, deviation = 0.0	Validation succeeded ytotal = 75.02484945753517, deviation = 6.394884621840902E-13

Figure 17. Output for multi-threaded programs, aspect-oriented versions versus non-aspect-oriented versions

For the Series programs, in calculating Fourier coefficients, the aspect-oriented and non-aspect-oriented programs generated the same results. Figure 17 indicates that the calculated Fourier coefficients are correct by showing the same "computed value" and "reference value" for each coefficient.

For the Crypt programs, in performing encryption and decryption, the aspect-oriented and non-aspect-oriented programs generated the same results. The four byte values shown in Figure 17 correspond to the first bytes encrypted and decrypted for each of the programs' four threads. Figure 17 indicates that the encryptions and decryptions were done correctly – the figure shows the same "original byte" and "decrypted byte" values for each of the four pairs of values. The order of the pair values is different due to the multi-threaded nature of the programs.

For the SOR programs, the aspect-oriented and the non-aspect-oriented programs calculated almost the same linear equation solution. The aspect-oriented program calculated a more accurate solution, with a deviation of 0 from the known reference solution, as opposed to the non-aspect-oriented program, which calculated a solution with a deviation of 2.313×10^{-6} from the known reference solution. But the aspect-oriented program's more accurate solution is misleading, in that the parallelizable loop pointcut was unable to match any of the loops in the aspect-oriented program, so the program ran without multi-threading. As explained earlier, the SOR algorithm is inherently sequential in nature, so running the program without multi-threading led to the more accurate solution.

For the SparseMatMult programs, the aspect-oriented and the non-aspect-oriented programs calculated almost the same matrix multiplication solution. The aspect-oriented

program calculated a more accurate solution, with a deviation of 0 from the known reference solution, as opposed to the non-aspect-oriented program, which calculated a solution with a deviation of 6.395×10^{-13} from the known reference solution. But, as with the SOR programs, the SparseMatMult aspect-oriented program's more accurate solution is misleading, in that the parallelizable loop pointcut was unable to match any of the loops in the aspect-oriented program, so the program ran without multi-threading. As explained earlier, the SparseMatMult algorithm's multi-threaded partitioning scheme can introduce small errors into the calculation. Thus, running the program without multi-threading led to the more accurate solution.

To fully understand how the programs' outputs were generated, the reader is encouraged to examine the aspect-oriented test programs – Series, Crypt, SOR, SparseMatMult, and the aspect file that worked in conjunction with all four benchmark programs. Those programs can be found in Appendices A, B, C, D, and E.

Verification of Matching Parallelizable Loop Pointcut with Parallelizable Loops

This section examines whether the new parallelizable loop pointcut was able to match loops that were deemed parallelizable by manual analysis. Figure 18 shows the result of such an examination for each of the four aspect-oriented benchmark programs.

	Manual analysis indicates that the loop is parallelizable (y/n)	Trace output that indicates whether the loop executed in a parallel manner
Series	yes	min = 1 min = 2 min = 3 min = 4 (1 target loop: 4 loop starting points for 4 different threads)
Crypt	yes	min2 = 1 min2 = 0 min2 = 2 min2 = 3 (1 target loop: 4 loop starting points for 4 different threads)
SOR	no no	min1 = 0 min2 = 1 (2 target loops: 1 loop starting point for each single-thread loop)
SparseMatMult	no no	min1 = 0 min2 = 0 (2 target loops: 1 loop starting point for each single-thread loop)

Figure 18. Comparison of manual analysis of loop parallelizability and application of a parallelizable loop pointcut

In Figure 18, note that for each loop targeted for parallelization, if the manual analysis indicates "yes" for parallelizable, then the trace output indicates that the loop was parallelized by matching the aspect-oriented compiler's parallelizable loop pointcut. For example, the Series program has one loop targeted for parallelization (i.e., the Java Grande Forum's multi-threaded Series program uses thread code for one loop). For the targeted loop in the aspect-oriented Series program, manual analysis indicated "yes" for parallelizability, and the output displayed four different starting values for the loop, one starting value for each of the four threads: min = 1, min = 2, min = 3, and min = 4. As explained in the Methodology chapter, this study's aspect-oriented advice parallelizes loops by interleaving the loop starting points for each thread and using a loop step value

equal to the number of threads. So for the Series program, which uses four threads and uses i = 1 for its first loop iteration, the first thread's loop executes iterations 1, 5, 9, …., the second thread's loop executes iterations 2, 6, 10, …, the third thread's loop executes iterations 3, 7, 11, …, and the fourth thread's loop executes iterations 4, 8, 12, ….

In Figure 18, note that for each loop targeted for parallelization, if the manual analysis indicates "no" for parallelizable, then the trace output indicates that the loop was not parallelized. For example, the SOR program has two loops targeted for parallelization. For both of the targeted loops in the aspect-oriented Series program, manual analysis indicated "no" for parallelizability, and the output displayed just one starting value for each of the two loops: min1 = 0, min2 = 1. One starting value for each loop means that only one thread was used for each loop.

To fully understand how the programs' outputs were generated, the reader is encouraged to examine the aspect-oriented test programs – Series, Crypt, SOR, SparseMatMult, and the aspect file that worked in conjunction with all four benchmark programs. Those programs can be found in Appendices A, B, C, D, and E.

Loop Parallelizability Analysis for the Benchmark Programs

The Java Grande Forum Benchmark Suite of programs contains versions of its programs that are implemented as parallel programs using threads. Although those programs contain loops that are executed in parallel, the comparable loops in the aspect-oriented versions of those programs are not necessarily parallelizable. That's because in creating the aspect-oriented versions of those programs, it was necessary to refactor the programs in a prescribed manner so that they might be able to match the parallelizable

loop pointcut. The Methodology chapter described the prescribed refactoring process. Sometimes the prescribed refactoring process led to loops that were no longer parallelizable. That was the case for the SOR and SparsMatMult programs, as evidenced by the no-parallelization results shown in Figure 18.

Figure 19 shows three loops from the SOR program that were not matched by the parallelizable loop pointcut. To avoid distracting code clutter, some of the original code, such as print statements, was removed from the figure. The first and second loops were not matched by the parallelizable loop pointcut because they both violate constraint 6 from the parallelizable loop detection algorithm presented earlier. Constraint 6 says that for each array reference definition, the array reference's index expression must include the loop's index variable. The first loop index variable is `p`, the second loop index variable is `j` and the loop contains an array reference that defines `Gi[j]`. Since the array reference's index expression `j` does not include the `p` loop index variable or the `i` loop index variable, constraint 6 is violated for both loops.

```
for (int p=0; p<num_iterations; p++)
{
  for (int i=1; i<Mm1; i++)
  {
    double [] Gi = G[i];
    double [] Gim1 = G[i-1];
    double [] Gip1 = G[i+1];

    for (int j=1; j<Nm1; j++)
    {
      Gi[j] = omega_over_four *
        (Gim1[j] + Gip1[j] + Gi[j-1] + Gi[j+1]) +
         one_minus_omega * Gi[j];
    }
  }
}
```

Figure 19. Loops in the SOR program that the aspect-oriented compiler could not parallelize

The third loop in Figure 19 was not matched by the parallelizable loop pointcut because it violates constraint 5 from the parallelizable loop detection algorithm. Constraint 5 says that there must not be any array dependencies between different elements of an array. That constraint is violated by `Gi[j]` appearing on the left side of an assignment statement and `Gi[j-1]` appearing on the right side.

Figure 20 shows two additional loops from the SOR program that were not matched by the parallelizable loop pointcut. The loops were not matched by the parallelizable loop pointcut because they both violate constraint 2 from the parallelizable loop detection algorithm. Constraint 2 limits the uses and definitions for local scalar variables. In particular, Constraint 2 requires that for every use of a local scalar variable, the variable gets assigned its value either from earlier in the loop's execution or from before the loop was executed. In executing the compound assignment operation, `Gtotal += G[i][j]`, `GTotal` first gets used (it gets added to `G[i][j]`), and then it gets defined (it gets

assigned an updated value). By defining `Gtotal` after it is used, that's a violation of constraint 2.

```
for (int i=min2; i<max2; i+=step2)
{
  for (int j=1; j<Nm1; j++)
  {
    Gtotal += G[i][j];
  }
}
```

Figure 20. Additional loops in the SOR program that the aspect-oriented compiler could not parallelize

Figure 21 shows three loops from the SparseMatMult program that were not matched by the parallelizable loop pointcut. All three loops were not matched by the parallelizable loop pointcut because they violate constraint 2 from the parallelizable loop detection algorithm. In the figure, note the += compound assignment statement with the variable `y[row[i]]` at its left. When the aspect-oriented compiler processes a compound assignment statement, it generates a Jimple use statement for the variable at the left, followed by a Jimple definition statement for that same variable. Such a use-definition construct violates constraint 2, which limits the uses and definitions for local scalar variables.

```
for (int reps=0; reps< SPARSE_NUM_ITER; reps++)
{
  for (int i=0; i<nz; i++)
  {
    y[row[i]] += x[col[i]] * val[i];
  }
}
.
.
.
for (int i=0; i<nz; i++)
{
  ytotal += y[row[i]];
}
```

Figure 21. Loops in the SparseMatMult program that the aspect-oriented compiler could not parallelize

Overcoming Problems Inherent in the Preexisting Software

In trying to implement a parallelizable loop pointcut for an aspect-oriented compiler, a great deal of effort was required to overcome problems inherent in the preexisting software that this study used as a starting point. The problems involved behavior that was different from what this study's researcher expected. The preexisting software was created primarily for research purposes, and, as such, documentation was relatively sparse, and the unexpected behavior wasn't addressed in the documentation.

As mentioned earlier, Sadat-Mohtasham's `transcut` pointcut was designed to group pointcuts so they could be matched with a group of join points (Sadat-Mohtasham, 2011). The transcut tool's documentation described using an `args` pointcut in the normal aspect-oriented manner – in conjunction with other pointcuts that relied on argument values. For example, traditional aspect-oriented compilers use `args` pointcuts with (1) method and constructor calls to match their argument values, (2) exception handling code to match thrown exception objects, and (3) field assignments to match their assigned

values. The following code shows how a `transcut` pointcut successfully matches a method call (`Math.sqrt`) and the method call's double argument value (`x`):

```
transcut tc(double x)
{
  pointcut test:
    call (double Math.sqrt(double)) && args(x);
}
```

With that example in mind, this study's researcher used the transcut construct to group each of the three loop pointcuts (`looped`, `loopPar`, and `outerLoopPar`) with `args` pointcuts, but those attempts did not work. Specifically, the code shown in Figure 22 did not work. The researcher tried to get that code to work by modifying the transcut tool's underlying software, but failed in that attempt. The workaround was to move `&& args` out of the transcut construct and into the advice construct, as shown in Figure 16.

```
public aspect LoopParTestAspect
{
  transcut tc(min, max, step)
  {
    pointcut loop: loopPar() && args(min, max, step);
  }

  void around(int min, int max, int step):
    tc()
  {
    .
    .
    .
  } // end around
} // end aspect LoopParTestAspect
```

Figure 22. This pairing of a loop pointcut and an `args` pointcut does not work

Another unexpected problem was the inability to weave new `args` values into a targeted loop's `min`, `max`, and `step` values if those values are constants. For example, in

the following loop heading, if there was an attempt to use the transcut and abc weaver to replace 0, 100, and 1 values with distinct values for separate threads, it wouldn't work.

```
for (int i=0; i<100; i++)
```

The transcut and abc weaver software requires replaced values to be variables, not constants. The researcher tried to get constants to work by modifying the transcut and abc software, but failed in that attempt. The workaround was to refactor the targeted loops by assigning the constant values to variables above each loop and using the variables in the loop heading. For example:

```
int min = 0;
int max = 100;
int step = 1;
for (int i=min; i<max; i+=step)
```

Another unexpected problem was that even after replacing loop heading constant values with variables, the min value did not get woven properly by the weaver. The min value didn't get woven properly because the compiler generated Jimple bytecode with the loop's index initialization statement appearing before the loop (which makes sense, since the initialization should not be repeated). In order to weave a value into the initialization statement (which is necessary for multi-threading), it was necessary to adjust the starting point for the loop's weaving process. The starting point is controlled by the appearance of a Jimple NOP (no operation) statement, so the solution involved locating the loop's initialization statement and inserting a NOP statement before it.

Software Repository

The software used in this study can be found at http://captain.park.edu/jdean/nova/dissertation/parallelizableLoopAspectsSoftware.html. That web page provides a downloadable zip file that contains (1) the soot jar file that was available from the Sable research group at the time of this study's implementation efforts, (2) abc jar files generated by the project's build file, and (3) source code that was added to soot and abc to form the parallelizable loop pointcuts and their matching and weaving mechanisms. Originally, that added code came from the transcut study. The current study made modifications to that code and added new files. The following list shows the files that contain most of the new code and brief descriptions of those files.

HashMutablePDG.java –
: Implements a program dependency graph for a given program. It uses the program dependency graph (PDG) to identify loops that are parallelizable.

PDGTranscutMatcher.java –
: Uses the PDG to match pointcuts, including the new parallelizable loop pointcuts, to a program's join points.

RegionShadowMatch.java –
: Uses the PDG to match pointcuts in a given region. Implements the ability to retrieve args values for a loop join point.

PDGRegion.java –
: Implements the functionality of a region in the program's PDG.

LoopedPDGNode.java –
: Implements a node in the PDG such that the node indicates the presence of a loop.

LoopParPointcut.java –
: Implements the new parallelizable loop pointcut that matches parallelizable loops regardless of whether they are inside other loops.

OuterLoopParPointcut.java –
Implements the new parallelizable loop pointcut that matches parallelizable loops that are not inside other loops.

contextcut.parser –
Defines the syntax for the parallelizable loop pointcuts.

contextcut.ast –
Defines the parallelizable loop pointcut expressions for the abc compiler's abstract syntax tree.

LoopParTest_SeriesWithoutAOP.java, LoopParTest_IDEAWithoutAOP.java, LoopParTest_SORWithoutAOP.java, LoopParTest_SparseMatMultWithoutAOP.java –
Java Grande Forum benchmark suite of programs with embedded multi-threading code that parallelizes the parallelizable loops.

LoopParTest_Series.java, LoopParTest_IDEA.java, LoopParTest_SOR.java, LoopParTest_SparseMatMult.java –
Java Grande Forum benchmark suite of programs that relies on an aspect file and this study's aspect-oriented compiler to parallelize the parallelizable loops.

LoopParTest.java –
An aspect file that matches parallelizable loops and replaces such loops with multi-threading code that parallelizes the matched loops. In addition to the aspect code, this file also contains Java source code with numerous loops used for incremental testing the parallelizable loop detection algorithm.

Summary

The Results chapter started by describing each of the six constraints in this study's parallelizable loop detection algorithm. The chapter then explained how transcut pointcuts and parallelizable loop pointcuts are used within an aspect file in order to check for parallelizable loops and weave the file's multi-threading advice code into the core concerns of target Java programs. The chapter then described each of the four benchmark programs in terms of their general purpose. The chapter then showed that for each benchmark pair of programs, the aspect-oriented version produced the same or nearly the same results as the non-aspect-oriented version. Next, the chapter provided results that

showed that the new parallelizable loop pointcut was able to identify parallelizable loops in two of the four benchmark programs and weave multi-threading advice into the loops identified as parallelizable. That indicated that the multi-threading advice was applied correctly. For two of the benchmark programs, the parallelizable loop pointcut was unable to identify parallelizable loops, and the chapter explained why the parallelizable loop pointcut was unable to identify those parallelizable loops as parallelizable. Next, the chapter addressed several unexpected problems that arose due to problems inherent in the preexisting software that this study's solution relies on. The chapter concluded by providing a description of the study's software repository.

Chapter 5

Conclusions, Implications, Recommendations, and Summary

Conclusions

This study's primary goal was to implement a parallelizable loop pointcut, The study's primary question was whether the parallelizable loop pointcut could be implemented correctly. The previous chapter's results show that this study was able to implement such a loop pointcut correctly.

The study's stated goals were to (1) define a pointcut for loops that are safely parallelizable, (2) implement the defined parallelizable loop pointcut, and (3) modify an existing aspect-oriented compiler so that its matching and weaving mechanisms worked with the new parallelizable loop pointcut. The first goal, defining a parallelizable loop pointcut, was accomplished fully, with two loop pointcuts defined – `loopPar`, for matching all parallelizable loops, and `outerLoopPar`, for matching all parallelizable loops that are not inside another loop. The second goal, implementing a parallelizable loop pointcut, was accomplished fully for both the `loopPar` and `outerLoopPar` pointcuts. The third goal, matching and weaving successfully with the new parallelizable loop pointcut, was accomplished, but to get the weaving process to work, refactoring target loops was necessary. Specifically, each target loop heading needs to use variables, not constants, for its min, max, and step values. And the variables need to be assigned their constant values above each loop.

A limitation of the study's resulting aspect-oriented compiler is the heuristic nature of its parallelizable loop detection algorithm. If the algorithm identifies a loop as parallelizable, the study found that the loop was indeed parallelizable. But for some parallelizable loops, the algorithm is unable to identify them as being parallelizable. This limitation is due to the difficulty of solving the parallelizable-loop-detection problem completely. Determining whether a loop is parallelizable is known to be an NP-complete problem (Aho, Lam, Sethi, & Ullman, 2007; Kyriakopoulos & Psarris, 2004), so using a heuristic algorithm is reasonable and practical.

Theoretically, this limitation of not being able to recognize some parallelizable loops as being parallelizable could be overcome by making the algorithm's constraints less conservative. Here are the constraints for identifying a parallelizable loop:

1. The loop must be trivial.
2. Limited uses and definitions for local scalar variables.
3. No field modifications.
4. Safe method calls only.
5. No array dependencies between different elements of an array.
6. For each array reference definition, the array reference's index expression must include the loop's index variable.

An example of how to make the algorithm more aggressive in terms of recognizing parallelizability for all parallelizable loops would be to omit constraint 3 and allow field modifications within target loops. But as explained earlier, allowing field modifications would require points-to analysis to ensure that the field modifications were safe for the

parallelization process. And points-to analysis can increase the algorithm's complexity significantly (Wu et al., 2002).

Implications

This study contributed to the discipline of computer science by introducing parallelizable loop pointcuts to an aspect-oriented compiler. Prior to this study, several prototype solutions existed for loop pointcuts, but the solutions were relatively coarse. In particular, they were unable to differentiate between loops that are parallelizable and those that are not. In creating a loop pointcut that is able to identify loops that are parallelizable, this study's pointcut should enable programmers to reduce the amount of time spent identifying loops that are parallelizable.

Being able to identify parallelizable loops automatically, as part of an aspect-oriented compiler's matching and weaving processes, is particularly important because (1) manually identifying parallelizable loops is known to be a difficult problem and (2) aspectizing parallelized loops can lead to a reduction in code tangling and an increase in separation of concerns.

By implementing an aspect pointcut that targets parallelizable loops, this study should make it easier for programmers to introduce safe parallelization to their programs and therefore should make it more likely that such parallelization will occur. The primary benefit of parallelizing loops is that it can lead to programs that execute faster. This speed-up is particularly evident when scientific computation "number crunching" is involved.

The study's researcher had some difficulty finding enough useful, real-world programs that could be parallelized by the study's parallelizable loop pointcut aspect-oriented compiler. Two of the four chosen test programs from the Java Grande Forum Benchmark Suite were parallelized manually, but the loops were not recognized as parallelizable by the parallelizable loop pointcut. Theoretically, it should have been possible to refactor the programs so they use loops that are matched by the parallelizable loop pointcut. But such refactoring would defeat one of the main points of aspect-oriented programming, which is to allow programs to exist in their natural form, without having to refactor them to accommodate concerns that are different from the programs' primary concerns.

Recommendations

This study has reinforced the notion that abc, soot, and Jimple are useful tools for programming language and compiler research efforts. The abc compiler relies on soot for its underlying backend software. Soot converts back and forth between Java source code and Jimple and between Jimple and bytecode. Jimple was designed to overcome the inherent difficulty of analyzing and working with Java bytecode. One thing in particular that makes bytecode difficult to work with is its reliance on the stack to hold intermediate values. Jimple uses a three-address instruction set with no reliance on a stack, which makes a program's flow of data easier to analyze (Lam et al., 2011).

A drawback to using abc, soot, and Jimple is that their documentation is somewhat sparse. The best resources for help are the soot and Jimple API libraries and the soot and abc forums.

As noted earlier, a limitation of this study's parallelizable loop pointcut aspect-oriented compiler is that its parallelizable loop detection algorithm is conservative in its attempt to identify parallelizable loops. In particular, the array-dependency test is conservative. For example, it doesn't allow for the possibility of skipping array accesses (e.g., odd and even array element accesses) (Aho et al., 2007).

To improve the data dependence analysis, and make it more aggressive in terms of identifying parallelizable loops, future studies could use Aho, et al.'s data dependence analysis algorithm (2007), which is based on the work of (Maydan, Hennessy, & Lam, 1991). The data dependence analysis algorithm was designed to solve a set of equations that define data dependencies between iterations of a loop. If the equations' solution finds no data dependencies, then the loop is deemed safe for parallelization. The most straightforward way to solve the set of equations is with integer linear programming, but integer linear programming is an NP-complete problem (Aho et al., 2007). With that in mind, the data dependence analysis algorithm relies on heuristic techniques to solve the equations.

The set of equations consists of equalities as well as inequalities. The data dependence analysis algorithm's first step is to use the Greatest Common Divisor (GCD) test to check for an integer solution to the equality equations. If there is no such solution, then there are no data dependencies. If there is a solution, the solved-for values are plugged into the inequality equations, which make the inequalities easier to solve.

The data dependence analysis algorithm's second step is to use a battery of heuristic tests that work together to determine whether there is a solution to the inequality

equations. The heuristic tests are the independent-variables test, the acyclic test, the loop-residue test, and the memorization test, and they are described in (Aho et al., 2007).

Summary

The primary goal of this study was to implement a parallelizable loop pointcut for an aspect-oriented compiler. To achieve that goal, the researcher designed and implemented an algorithm that determines whether a given loop is parallelizable. Determining whether a loop is parallelizable is known to be a very difficult problem (Aho et al., 2007; Kyriakopoulos & Psarris, 2004), so much of this study's effort focused on that issue. A secondary goal of this study was to modify an existing aspect-oriented compiler so that its matching and weaving mechanisms worked with the new parallelizable loop pointcut.

Being able to aspectize the parallelization of loops is particularly important because loop parallelization code is a heterogeneous concern, and heterogeneous concerns tend to have tangled code (Trifu & Kuttruff, 2005). By aspectizing the parallelization of loops, this study's parallelizable loop pointcut has made it easier to reduce code tangling with loop parallelization code. And a reduction in code tangling can lead to improvements in maintenance, debugging, and code reuse (Eaddy & Aho, 2006).

By implementing an aspect pointcut that targets parallelizable loops, this study has made it easier for programmers to introduce safe parallelization to their programs, and therefore should make it more likely that such parallelization occurs. The benefit of such parallelization is made clear by *Amdahl's Law*, which states that the execution speed of a parallelized program is a function of *f*, the fraction of the program's executed code that is

parallelized, and p, the number of processors employed to run the program (Aho et al., 2007):

$$\frac{1}{(1-f)+(f/p)}$$

The $1-f$ term dominates the denominator, so as f increases, the speed increases as well. To a lesser degree, as p increases, the speed increases. According to (Aho et al., 2007), programs typically spend most of their time executing loops, so improving the performance of loops can have a significant impact on the overall performance of programs.

This study's focal construct, the loop, is more difficult to map to a pointcut(s) than other constructs because loops don't have standard arguments (like method arguments) or identifiers that can be used to identify them (Harbulot & Gurd, 2006). If pointcut join points were identified at the source code level, then there would be a need for different pointcuts for while, do, and for loops. In that case, programmers' personal preferences for loop types would impact which pointcuts were used, and the types of pointcuts used could impact the effectiveness of a particular program's aspectization (Harbulot & Gurd, 2006). To avoid such programmer-specific variability, this study identified pointcut join points at the bytecode level, rather than at the source code level.

This study's literature review found only a few studies that implemented loop pointcuts that matched at the bytecode level. One such example was Harbulot and Gurd's LoopsAJ aspect-oriented compiler. It was particularly appealing as a possible starting point for this study because LoopsAJ differentiates between loops with different numbers of exit points, and that differentiation is known to be helpful when identifying loops that are parallelizable (Bik & Gannon, 1997; Harbulot & Gurd, 2006).

Another study that implemented loop pointcuts that matched at the bytecode level was the transcut study, conducted by (Sadat-Mohtasham, 2011). The current study found that the transcut tool was better to work with than the LoopsAJ tool. Transcut was built using abc, Soot, and Jimple. Abc is an aspect-oriented compiler whose hallmark is extensibility. Soot is a well-established software framework tool. Abc uses Soot to implement abc's backend. The backend provides the analysis and code generation necessary for the matching and weaving operations inherent in abc's aspect-oriented functionality. To help with the analysis and manipulation of Java programs, Soot converts Java target programs to Jimple code. Jimple has an instruction set that is in between high-level Java source code and low-level bytecode in terms of its complexity.

The transcut tool (and, consequently, this study) relies on a program dependency graph (PDG) to represent a program's control dependencies, using nodes for program instructions and regions and edges for connections between them. With a PDG, a connection arrow goes from node A to node B if node B is dependent on node A. Such connections and dependencies are particularly important for identifying and analyzing loops.

The study's primary question was whether a parallelizable loop pointcut could be implemented correctly. This study provided results that showed that such a loop pointcut was implemented correctly. Specifically, the study showed that for a set of target benchmark programs, the parallelizable-loop-pointcut aspect-oriented compiler produced the same or nearly the same results as a non-aspect-oriented compiler. Also, the study showed that for the same set of target benchmark programs, the parallelizable-loop-pointcut aspect-oriented compiler was able to identify parallelizable loops in two of the

four benchmark programs and weave multi-threading advice into the loops identified as parallelizable. The inability to identify the other two benchmark programs as parallelizable was due to the difficulty of solving the parallelizable-loop-detection problem completely.

As part of the implementation process for the parallelizable loop pointcut, the study achieved a secondary outcome – the design and implementation of an algorithm that detects whether a loop is parallelizable. Such a detection process is known to be a difficult problem, so the algorithm was heuristic in nature. Another secondary outcome of this study was the determination of how transcut pointcuts and parallelizable loop pointcuts can be used within an aspect file to check for parallelizable loops and weave the file's multi-threading advice code into the core concerns of target Java programs. This study provided an example aspect which illustrated that pointcut usage.

Appendix A

Benchmark File LoopParTest_Series.java – a Target for Aspect-Oriented Parallelization

The following code is from the LoopParTest_Series.java file. It relies on an aspect file and this study's aspect-oriented compiler to parallelize the parallelizable loops. The non-aspect-oriented version of the file came from the Java Grande Forum benchmark suite of programs.

```
/** This class runs the Series program using threads with AOP.
 *
 *  @author John Dean
 *  March 3, 2013
 */

import java.util.Random;

public class LoopParTest_Series
{
  public static void main(String[] args)
  {
    int arrayRows = 10000;
    double[][] testArray;

    testArray = new double[2][arrayRows];

    double omega;       // Fundamental frequency.

    // Calculate the fourier series. Begin by calculating A[0].
    testArray[0][0] =
      trapezoidIntegrate(
        (double) 0.0, (double) 2.0, 1000, (double) 0.0, 0) /
          (double) 2.0;

    // Calculate the fundamental frequency.
    // ( 2 * pi ) / period...and since the period
    // is 2, omega is simply pi.

    omega = (double) 3.1415926535897932;

    int min = 1;
    int max = arrayRows;
    int step = 1;
```

```
  for (int i=min; i<max; i+=step)
  {
    if (i <= min)
    {
     System.out.println("min = " + min);
      System.out.println("max = " + max);
      System.out.println("step = " + step);
    }

    // Calculate A[i] terms. Note, once again, that we
    // can ignore the 2/period term outside the integral
    // since the period is 2 and the term cancels itself
    // out.

    // 1 = cosine term, 2 = sine term
    testArray[0][i] =
      trapezoidIntegrate(
        (double) 0.0, (double) 2.0, 1000, omega * (double) i, 1);
    testArray[1][i] =
      trapezoidIntegrate(
        (double) 0.0, (double) 2.0, 1000, omega * (double) i, 2);
  } // end for

  double ref[][] =
    {{2.8729524964837996, 0.0},
     {1.1161046676147888, -1.8819691893398025},
     {0.34429060398168704, -1.1645642623320958},
     {0.15238898702519288, -0.8143461113044298}};

  for (int i=0; i<4; i++)
  {
    for (int j=0; j<2; j++)
    {
     double error = Math.abs(testArray[j][i] - ref[i][j]);
     if (error > 1.0e-12)
     {
       System.out.println("Validation failed");
     }
     else
     {
        System.out.println("Validation succeeded");
     }

     System.out.println("coefficient " + j + "," + i);
     System.out.println("Computed value = " + testArray[j][i]);
     System.out.println("Reference value = " + ref[i][j]);
    } // end for
  } // end for
} // end main

//*************************************************************

// This method performs a simple trapezoid integration on the
// function (x+1)**x.
// x0,x1 set the lower and upper bounds of the integration.
// nsteps indicates # of trapezoidal sections.
```

```
// omegan is the fundamental frequency times the series member #.
// select = 0 for the A[0] term, 1 for cosine terms, and 2 for
// sine terms. Returns the value.

private static double trapezoidIntegrate(
  double x0, double x1, int nsteps, double omegan, int select)
{
  double x;               // Independent variable.
  double dx;              // Step size.
  double rvalue;          // Return value.
  double fn = 0.0;        // the function

  // Initialize independent variable.
  x = x0;

  // Calculate stepsize.
  dx = (x1 - x0) / (double) nsteps;

  // Initialize the return value.
  switch (select)
  {
    case 0:
     fn = Math.pow(x0 + 1.0, x0); break;
    case 1:
     fn = Math.pow(x0 + 1.0, x0) * Math.cos(omegan * x0); break;
    case 2:
     fn = Math.pow(x0 + 1.0, x0) * Math.sin(omegan * x0); break;
    default:
     fn = 0.0;
  }

  rvalue = fn / (double) 2.0;

  // Compute the other terms of the integral.
  if (nsteps != 1)
  {
    --nsteps;               // Already done 1 step.
    while (--nsteps > 0)
    {
      x += dx;
      switch (select)
      {
        case 0:
         fn = Math.pow(x + 1.0, x); break;
        case 1:
         fn = Math.pow(x + 1.0, x) * Math.cos(omegan * x); break;
        case 2:
         fn = Math.pow(x + 1.0, x) * Math.sin(omegan * x); break;
        default:
         fn = 0.0;
      }
      rvalue += fn;
    }
  }

  switch (select)
  {
```

```
      case 0:
       fn = Math.pow(x1 + 1.0, x1); break;
      case 1:
       fn = Math.pow(x1 + 1.0, x1) * Math.cos(omegan * x1); break;
      case 2:
       fn = Math.pow(x1 + 1.0, x1) * Math.sin(omegan * x1); break;
      default:
       fn = 0.0;
    }

    rvalue = (rvalue + fn / 2.0) * dx;
    return rvalue;
  } // end trapezoidIntegrate
} // end class LoopParTest_Series
```

Appendix B

Benchmark File LoopParTest_IDEA.java – a Target for Aspect-Oriented Parallelization

The following code is from the LoopParTest_IDEA.java file. It relies on an aspect file and this study's aspect-oriented compiler to parallelize the parallelizable loops. The non-aspect-oriented version of the file came from the Java Grande Forum benchmark suite of programs.

```
/** This class runs the IDEA encryption and decryption program using
 * threads with AOP.
 *
 *  @author John Dean
 *  March 9, 2013
 */

import java.util.*;

class LoopParTest_IDEA
{
  int array_rows;

  byte[] plain1;       // Buffer for plaintext data.
  byte[] crypt1;       // Buffer for encrypted data.
  byte[] plain2;       // Buffer for decrypted data.

  short[] userkey;     // Key for encryption/decryption.
  int[] Z;             // Encryption subkey (userkey derived).
  int[] DK;            // Decryption subkey (userkey derived).

  public static void main(String[] args)
  {
    LoopParTest_IDEA test;

    test = new LoopParTest_IDEA();
    test.run();
  } // end main

  public void run()
  {
    array_rows = 10000;
   buildTestData();
   Do();
```

```
  int min2 = 0;
   int max2 = array_rows;
   int step2 = 1;

   for (int i=min2; i<max2; i+=step2)
   {
     if (i <= min2)
     {
       System.out.println("min2 = " + min2);
       System.out.println("max2 = " + max2);
       System.out.println("step2 = " + step2);
     }
     if (plain1[i] != plain2[i])
     {
       System.out.println("Validation failed");
       System.out.println("Original Byte " + i + " = " + plain1[i]);
       System.out.println("Encrypted Byte " + i + " = " + crypt1[i]);
       System.out.println("Decrypted Byte " + i + " = " + plain2[i]);
     }

     // If the 1st byte is a success, display it.
     else if (i <= min2)
     {
      System.out.println("1st original byte: " + plain1[i]);
      System.out.println("1st encrypted byte: " + crypt1[i]);
      System.out.println("1st decrypted byte: " + plain2[i]);
     }
   } // end for

   System.out.println(
     "IDEA encryption/decryption finished.\n" +
     " If no \"validation failed\" messages displayed," +
     " then validation succeeded.");
} // end run

void Do()
{
  cipher_idea(plain1, crypt1, Z);     // Encrypt plain1.
   cipher_idea(crypt1, plain2, DK);    // Decrypt.
}

void buildTestData()
{
   plain1 = new byte [array_rows];
   crypt1 = new byte [array_rows];
   plain2 = new byte [array_rows];

   Random rndnum = new Random(136506717L);

   userkey = new short [8];  // User key has 8 16-bit shorts.
   Z = new int [52];         // Encryption subkey (user key derived).
   DK = new int [52];        // Decryption subkey (user key derived).

   for (int i = 0; i < 8; i++)
   {
     userkey[i] = (short) rndnum.nextInt();
```

```
  }

  calcEncryptKey();
  calcDecryptKey();

  // Fill plain1 with "text."
  for (int i = 0; i < array_rows; i++)
  {
    plain1[i] = (byte) i;
  }
}

private void calcEncryptKey()
{
  int j;                        // Utility variable.

  for (int i = 0; i < 52; i++) // Zero out the 52-int Z array.
    Z[i] = 0;

  for (int i = 0; i < 8; i++)  // First 8 subkeys are userkey.
  {
    Z[i] = userkey[i] & 0xffff;     // Convert "unsigned"
                                      // short to int.
  }

  for (int i = 8; i < 52; i++)
  {
      j = i % 8;
      if (j < 6)
      {
          Z[i] = ((Z[i -7]>>>9) | (Z[i-6]<<7)) // Shift and combine
                  & 0xFFFF;                    // Just 16 bits.
          continue;                            // Next iteration.
      }

      if (j == 6)    // Wrap to beginning for second chunk.
      {
          Z[i] = ((Z[i -7]>>>9) | (Z[i-14]<<7))
                  & 0xFFFF;
          continue;
      }

       // j == 7 so wrap to beginning for both chunks.

      Z[i] = ((Z[i -15]>>>9) | (Z[i-14]<<7)) & 0xFFFF;
  }
}

private void calcDecryptKey()
{
  int j, k;                // Index counters.
  int t1, t2, t3;          // Temps to hold decrypt subkeys.

  t1 = inv(Z[0]);          // Multiplicative inverse (mod x10001).
  t2 = - Z[1] & 0xffff;    // Additive inverse, 2nd encrypt subkey.
  t3 = - Z[2] & 0xffff;    // Additive inverse, 3rd encrypt subkey.
```

```
  DK[51] = inv(Z[3]);      // Multiplicative inverse (mod x10001).
  DK[50] = t3;
  DK[49] = t2;
  DK[48] = t1;

  j = 47;                  // Indices into temp and encrypt arrays.
  k = 4;
  for (int i = 0; i < 7; i++)
  {
      t1 = Z[k++];
      DK[j--] = Z[k++];
      DK[j--] = t1;
      t1 = inv(Z[k++]);
      t2 = -Z[k++] & 0xffff;
      t3 = -Z[k++] & 0xffff;
      DK[j--] = inv(Z[k++]);
      DK[j--] = t2;
      DK[j--] = t3;
      DK[j--] = t1;
  }

  t1 = Z[k++];
  DK[j--] = Z[k++];
  DK[j--] = t1;
  t1 = inv(Z[k++]);
  t2 = -Z[k++] & 0xffff;
  t3 = -Z[k++] & 0xffff;
  DK[j--] = inv(Z[k++]);
  DK[j--] = t3;
  DK[j--] = t2;
  DK[j--] = t1;
}

//*************************************************************

private void cipher_idea(byte [] text1, byte [] text2, int [] key)
{
  int i1 = 0;                // Index into first text array.
  int i2 = 0;                // Index into second text array.
  int ik;                    // Index into key array.
  int x1, x2, x3, x4, t1, t2; // Four "16-bit" blocks, two temps.
  int r;                     // Eight rounds of processing.

  int min1 = 0;
  int max1 = text1.length;
  int step1 = 8;

  for (int i = min1; i < max1; i += step1)
  {
    if (i <= min1)
    {
      System.out.println("min1 = " + min1);
      System.out.println("max1 = " + max1);
      System.out.println("step1 = " + step1);
    }

    ik = 0;                  // Restart key index.
```

```
    r = 8;                  // Eight rounds of processing.

    // Load eight plain1 bytes as four 16-bit "unsigned" integers.
    // Masking with 0xff prevents sign extension with cast to int.

    // The following compound assignment operators prevent the
    // parallelizable loop pointcut from matching this loop.

    x1 = text1[i1++] & 0xff;         // 16-bit x1 from 2 bytes,
    x1 |= (text1[i1++] & 0xff) << 8; // assuming low-order byte 1st.
    x2 = text1[i1++] & 0xff;
    x2 |= (text1[i1++] & 0xff) << 8;
    x3 = text1[i1++] & 0xff;
    x3 |= (text1[i1++] & 0xff) << 8;
    x4 = text1[i1++] & 0xff;
    x4 |= (text1[i1++] & 0xff) << 8;

    do
    {
      x1 = (int) ((long) x1 * key[ik++] % 0x10001L & 0xffff);
      x2 = x2 + key[ik++] & 0xffff;
      x3 = x3 + key[ik++] & 0xffff;
      x4 = (int) ((long) x4 * key[ik++] % 0x10001L & 0xffff);

      t2 = x1 ^ x3;
      t2 = (int) ((long) t2 * key[ik++] % 0x10001L & 0xffff);
      t1 = t2 + (x2 ^ x4) & 0xffff;
      t1 = (int) ((long) t1 * key[ik++] % 0x10001L & 0xffff);
      t2 = t1 + t2 & 0xffff;
      x1 ^= t1;
      x4 ^= t2;
      t2 ^= x2;
      x2 = x3 ^ t1;
      x3 = t2;        // Results of x2 and x3 now swapped.
    } while(--r != 0);  // Repeats seven more rounds.

    x1 = (int) ((long) x1 * key[ik++] % 0x10001L & 0xffff);
    x3 = x3 + key[ik++] & 0xffff;
    x2 = x2 + key[ik++] & 0xffff;
    x4 = (int) ((long) x4 * key[ik++] % 0x10001L & 0xffff);

    text2[i2++] = (byte) x1;
    text2[i2++] = (byte) (x1 >>> 8);
    text2[i2++] = (byte) x3;            // x3 and x2 are switched
    text2[i2++] = (byte) (x3 >>> 8);    // only in name.
    text2[i2++] = (byte) x2;
    text2[i2++] = (byte) (x2 >>> 8);
    text2[i2++] = (byte) x4;
    text2[i2++] = (byte) (x4 >>> 8);
  }   // End for loop.
}   // End routine.

//************************************************************

private int mul(int a, int b) throws ArithmeticException
{
  long p;             // Large enough to catch 16-bit multiply
```

```
                        // without hitting sign bit.
    if (a != 0)
    {
      if(b != 0)
      {
        p = (long) a * b;
        b = (int) p & 0xFFFF;       // Lower 16 bits.
        a = (int) p >>> 16;         // Upper 16 bits.

        return (b - a + (b < a ? 1 : 0) & 0xFFFF);
      }
      else
        return ((1 - a) & 0xFFFF);  // If b = 0, then same as
                                    // 0x10001 - a.
    }
    else                            // If a = 0, then return
      return((1 - b) & 0xFFFF);     // same as 0x10001 - b.
  }

  //************************************************************

  private int inv(int x)
  {
    int t0, t1;
    int q, y;

    if (x <= 1)         // Assumes positive x.
      return(x);        // 0 and 1 are self-inverse.

    t1 = 0x10001 / x;   // (2**16+1)/x; x is >= 2, so fits 16 bits.
    y = 0x10001 % x;
    if (y == 1)
      return((1 - t1) & 0xFFFF);

    t0 = 1;
    do
    {
      q = x / y;
      x = x % y;
      t0 += q * t1;
      if (x == 1) return(t0);
      q = y / x;
      y = y % x;
      t1 += q * t0;
    } while (y != 1);

    return((1 - t1) & 0xFFFF);
  }
} // end LoopParTest_IDEA
```

Appendix C

Benchmark File LoopParTest_SOR.java – a Target for Aspect-Oriented Parallelization

The following code is from the LoopParTest_SOR.java file. It relies on an aspect file and this study's aspect-oriented compiler to parallelize the parallelizable loops. The non-aspect-oriented version of the file came from the Java Grande Forum benchmark suite of programs.

```
/** This class runs the SOR program using threads with AOP.
 *
 *  @author John Dean
 *  February 25, 2013
 */

import java.util.Random;

public class LoopParTest_SOR
{
  private static final int JACOBI_NUM_ITER = 100;
  private static final long RANDOM_SEED = 10101010;

  public static void main(String[] args)
  {
    double Gtotal = 0.0;

    double omega = 1.25;
    double[][] G = randomMatrix(1000, 1000, new Random(RANDOM_SEED));
    int num_iterations = JACOBI_NUM_ITER;

    int M = G.length;
    int N = G[0].length;

    double omega_over_four = omega * 0.25;
    double one_minus_omega = 1.0 - omega;

    // update interior points
    int Mm1 = M-1;
    int Nm1 = N-1;

    int min1 = 0;
    int max1 = num_iterations;
    int step1 = 1;
```

```
  for (int p=min1; p<max1; p+=step1)
  {
    if (p <= min1)
    {
     System.out.println("min1 = " + min1);
     System.out.println("max1 = " + max1);
     System.out.println("step1 = " + step1);
    }

    for (int i=1; i<Mm1; i++)
    {
     double [] Gi = G[i];
      double [] Gim1 = G[i-1];
      double [] Gip1 = G[i+1];

      for (int j=1; j<Nm1; j++)
      {
        Gi[j] = omega_over_four * (Gim1[j] + Gip1[j] + Gi[j-1] +
         Gi[j+1]) + one_minus_omega * Gi[j];
      }
    }
  }

  int min2 = 1;
  int max2 = Nm1;
  int step2 = 1;

  for (int i=min2; i<max2; i+=step2)
  {
    if (i <= min2)
    {
      System.out.println("min2 = " + min2);
      System.out.println("max2 = " + max2);
      System.out.println("step2 = " + step2);
    }

    for (int j=1; j<Nm1; j++)
    {
      Gtotal += G[i][j];
    }
  }

  double dev = Math.abs(Gtotal - 0.4984199298207158);

  if (dev > 1.0e-5 )
  {
    System.out.println("Validation failed");
  }
  else
  {
    System.out.println("Validation succeeded");
  }

  System.out.println(
    "Gtotal = " + Gtotal + ", deviation = " + dev);
} // end main
```

```
  //*************************************************************

  private static double[][] randomMatrix(int M, int N, Random r)
  {
    double[][] A = new double[M][N];

    for (int i=0; i<N; i++)
    {
      for (int j=0; j<N; j++)
      {
        A[i][j] = r.nextDouble() * 1e-6;
      }
    }
    return A;
  }
} // end class LoopParTest_SOR
```

Appendix D

Benchmark File LoopParTest_ SparseMatMult.java – a Target for Aspect-Oriented Parallelization

The following code is from the LoopParTest_SparseMatMult.java file. It relies on an aspect file and this study's aspect-oriented compiler to parallelize the parallelizable loops. The non-aspect-oriented version of the file came from the Java Grande Forum benchmark suite of programs.

```
/** This class runs the SparseMatMult program.
 * It was created for the purpose of attempting to use AOP to add
threads.
 *
 *  @author John Dean
 *  February 27, 2013
 */

import java.util.Random;

public class LoopParTest_SparseMatMult
{
  private double ytotal = 0.0;

  private int size;
  private static final long RANDOM_SEED = 10101010;

  private static final int datasizes_M[] = {50000,100000,500000};
  private static final int datasizes_N[] = {50000,100000,500000};
  private static final int datasizes_nz[] = {250000,500000,2500000};
  private static final int SPARSE_NUM_ITER = 200;

  Random r = new Random(RANDOM_SEED);

  double [] x;
  double [] y;
  double [] val;
  int [] col;
  int [] row;

  //*************************************************

  public void setSize(int size)
  {
```

```
  this.size = size;
}

//*******************************************************

public void initialize()
{
  x = randomVector(datasizes_N[size], r);
  y = new double[datasizes_M[size]];
  val = new double[datasizes_nz[size]];
  col = new int[datasizes_nz[size]];
  row = new int[datasizes_nz[size]];

  for (int i=0; i<datasizes_nz[size]; i++)
  {
    // generate random row index (0, M-1)
    row[i] = Math.abs(r.nextInt()) % datasizes_M[size];

    // generate random column index (0, N-1)
    col[i] = Math.abs(r.nextInt()) % datasizes_N[size];

    val[i] = r.nextDouble();
  }
} // end initialize

//*******************************************************

private static double[] randomVector(int N, Random r)
{
  double A[] = new double[N];

  for (int i=0; i<N; i++)
  {
    A[i] = r.nextDouble() * 1e-6;
  }

  return A;
} // end randomVector

//*******************************************************

public void test()
{
  int nz = val.length;

  int min1 = 0;
  int max1 = SPARSE_NUM_ITER;
  int step1 = 1;

  for (int reps=min1; reps<max1; reps+=step1)
  {
    if (reps <= min1)
    {
      System.out.println("min1 = " + min1);
      System.out.println("max1 = " + max1);
      System.out.println("step1 = " + step1);
    }
```

```
      for (int i=0; i<nz; i++)
      {
        y[row[i]] += x[col[i]] * val[i];
      }
    }

    int min2 = 0;
    int max2 = nz;
    int step2 = 1;

    for (int i=min2; i<max2; i+=step2)
    {
      if (i <= min2)
      {
        System.out.println("min2 = " + min2);
        System.out.println("max2 = " + max2);
        System.out.println("step2 = " + step2);
      }

      ytotal += y[row[i]];
    }
  } // end test

  //--------------------------------------------------

  public void validate()
  {
    double refval[] =
      {75.02484945753453,150.0130719633895,749.5245870753752};
    double dev = Math.abs(ytotal - refval[size]);

    if (dev > 1.0e-10)
    {
      System.out.println("Validation failed");
    }
    else
    {
      System.out.println("Validation succeeded");
    }

    System.out.println(
      "ytotal = " + ytotal + ", deviation = " + dev);
  } // end validate

  //--------------------------------------------------

  public static void main(String argv[])
  {
    LoopParTest_SparseMatMult smm = new LoopParTest_SparseMatMult();
    smm.setSize(0);
    smm.initialize();
    smm.test();
    smm.validate();
  }
} // end LoopParTest_SparseMatMult
```

Appendix E

LoopParTest.java – An Aspect File that Matches Parallelizable Loops

The following code is from the LoopParTest.java file. It contains an aspect that matches parallelizable loops and replaces such loops with multi-threading code that parallelizes the matched loops.

```
public class LoopParTest
{
  public static void main(String[] args)
  {
    int min = 0;
    int max = 10;
    int step = 1;

    for (int i=min; i<max; i+=step)
    {
      System.out.println("min = " + min);
      System.out.println("max = " + max);
      System.out.println("step = " + step);
    }
  } // end main
} // end class LoopParTest

//*************************************************

public aspect LoopParTestAspect
{
  transcut tc()
  {
    // pointcut loop: outerLoopPar();
    pointcut loop: loopPar();
  }

  void around(int min, int max, int step):
    tc() && args(min, max, step)
  {
    int numThreads = 4;
    Thread[] threads = new Thread[numThreads];

    for (int i=0; i<numThreads; i++)
    {
      // Compiler rule:
      // Within an inner class, if a local variable is used but not
      // declared, then it must be final. The scope of a variable
      // declared within a loop is a single loop iteration. Thus,
```

```
      // final variables' values can be re-assigned (see below).

      final int t_min = min + i;
      final int t_max = max;
      final int t_step = numThreads * step;

      Runnable r = new Runnable()
      {
        public void run()
        {
          proceed(t_min, t_max, t_step);
        }
      };
      threads[i] = new Thread(r);
    }

    for (int i=0; i<numThreads; i++)
    {
      threads[i].start();
    }

    try
    {
      for (int i=0; i<numThreads; i++)
      {
        threads[i].join();
      }
    }
    catch (InterruptedException e)
    { }
  } // end around
} // end aspect LoopParTestAspect
```

References

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers: Principles, Techniques and Tools* (2nd ed.). Boston: Addison-Wesley.

Avgustinov, P., Christensen, A., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., . . . Tibble, J. (2006). abc : An extensible aspectJ compiler. In A. Rashid & M. Aksit (Eds.), *Transactions on aspect-oriented software development I* (Vol. 3880, pp. 293-334). Berlin, Germany: Springer.

Banerjee, U., Eigenmann, R., Nicolau, A., & Padua, D. A. (1993). Automatic program parallelization. *Proceedings of the IEEE, 81*(2), 211-243. doi: 10.1109/5.214548

Bik, A. J., & Gannon, D. B. (1997). JAVAB - a prototype bytecode parallelization tool (pp. 1-46): Computer Science Department, Indiana University.

Bik, A. J., & Gannon, D. B. (1998). A prototype bytecode parallelization tool. *Concurrency: Practice and Experience, 10*(11-13), 879-885.

Bik, A. J., Villacis, J., & Gannon, D. B. (1998). Experiences with loop parallelization in javar (a prototype restructuring compiler for java). In Z. Li, P.-C. Yew, S. Chatterjee, C.-H. Huang, P. Sadayappan & D. Sehr (Eds.), *Languages and compilers for parallel computing* (pp. 355-366). Berlin, Germany: Springer.

Blume, W., Eigenmann, R., Hoeflinger, J., Padua, D., Petersen, P., Rauchwerger, L., & Tu, P. (1994). Automatic detection of parallelism: A grand challenge for high-performance computing. *IEEE Concurrency, 2*(3), 37-47. doi: 10.1109/M-PDT.1994.329796

Dig, D., Tarce, M., Radoi, C., Minea, M., & Johnson, R. (2009). Relooper: Refactoring for loop parallelism in Java. *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, 793-794. doi: 10.1145/1639950.1640018

Eaddy, M., & Aho, A. (2006). Statement annotations for fine-grained advising. *Proceedings of the Workshop on Reflection, AOP, and Meta-data for Software Evolution*, 89-100.

Einarsson, A., & Nielsen, J. D. (2008). *A survivor's guide to Java program analysis with Soot*. University of Aarhus, Denmark Retrieved from http://www.brics.dk/SootGuide/sootsurvivorsguide.pdf.

Ferrante, J., Ottenstein, K. J., & Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems, 9*(3), 319-349. doi: 10.1145/24039.24041

Garcia-Yaguez, A., Llanos, D., & Gonzalez-Escribano, A. (2013). Squashing alternatives for software-based speculative parallelization. *IEEE Transactions on Computers, PP*(99), 1-14. doi: 10.1109/TC.2013.46

Hammacher, C., Streit, K., Hack, S., & Zeller, A. (2009). Profiling Java programs for parallelism. *Proceedings of the ICSE Workshop on Multicore Software Engineering*, 49-55. doi: 10.1109/iwmse.2009.5071383

Harbulot, B. (2006). *Separating concerns in scientific software using aspect-oriented programming.* (Ph.D.), University of Manchester, Manchester.

Harbulot, B., & Gurd, J. R. (2004). Using AspectJ to separate concerns in parallel scientific Java code. *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, 122-131. doi: http://doi.acm.org/10.1145/976270.976286

Harbulot, B., & Gurd, J. R. (2006). A join point for loops in AspectJ. *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, 63-74. doi: http://doi.acm.org/10.1145/1119655.1119666

Hilsdale, E., & Hugunin, J. (2004). Advice weaving in AspectJ. *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, 26-35. doi: http://doi.acm.org/10.1145/976270.976276

Jenkov, J. (n.d.). Thread Safety and Shared Resources. http://tutorials.jenkov.com/java-concurrency/thread-safety.html

Judd, G., Clement, M., Snell, Q., & Getov, V. (1999). Design issues for efficient implementation of MPI in Java. *Proceedings of the ACM 1999 Conference on Java Grande*, 58-65. doi: http://doi.acm.org/10.1145/304065.304097

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., & Irwin, J. (1997). Aspect-oriented programming. *Proceedings of the European Conference on Object-Oriented Programming, 1241*, 220-242.

Kiczales, G., & Mezini, M. (2005). Separation of concerns with procedures, annotations, advice and pointcuts. *Proceedings of the 19th European Conference on Object-Oriented Programming, 3586*, 195-213.

Kniesel, G., & Austermann, M. (2002). Code coverage for Java: A load-time adaptation success story. In J. Bishop (Ed.), *Component Deployment* (Vol. 2370, pp. 155-169). Berlin, Germany: Springer.

Kyriakopoulos, K., & Psarris, K. (2004). Data dependence analysis techniques for increased accuracy and extracted parallelism. *International Journal of Parallel Programming, 32*(4), 317-359. doi: http://dx.doi.org/10.1023/B:IJPP.0000035817.01263.d0

Laddad, R. (2003). *AspectJ in Action*. Greenwich, CT: Manning Publications.

Lam, P., Bodden, E., Lhotak, O., & Hendren, L. (2011). The Soot framework for Java program analysis: a retrospective. *Proceedings of the Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*.

Leijen, D., & Hall, J. (2007). Parallel performance - optimize managed code for multi-core machines. *MSDN Magazine,* (October 2007). doi:http://msdn.microsoft.com/en-us/magazine/cc163340.aspx

Leijen, D., Schulte, W., & Burckhardt, S. (2009). *The design of a task parallel library*. Paper presented at the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, Orlando, Florida, USA.

Marron, M., Mendez-Lojo, M., Hermenegildo, M., Stefanovic, D., & Kapur, D. (2008). Sharing analysis of arrays, collections, and recursive structures. *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 43-49. doi: 10.1145/1512475.1512485

Maydan, D. E., Hennessy, J. L., & Lam, M. S. (1991). Efficient and exact data dependence analysis. *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, 1-14. doi: 10.1145/113445.113447

Microsoft. (2012). How to: Write a parallel_for loop. http://msdn.microsoft.com/en-us/library/dd728073.aspx

Psarris, K., & Kyriakopoulos, K. (2003). The impact of data dependence analysis on compilation and program parallelization. *Proceedings of the 17th Annual International Conference on Supercomputing*, 205-214. doi: http://doi.acm.org/10.1145/782814.782843

Rajan, H., & Sullivan, K. (2005). Generalizing AOP for aspect-oriented testing. *Proceedings of the Fourth International Conference on Aspect-Oriented Software Development*, 1-11.

Rho, T., Kniesel, G., & Appeltauer, M. (2006). Fine-grained generic aspects. *Proceedings of the Aspect Oriented Software Development Workshop on Foundations of Aspect-Oriented Languages*, 29-35.

Ricci, L. (2002). Automatic loop parallelization: An abstract interpretation approach. *Proceedings of the International Conference on Parallel Computing in Electrical Engineering*, 112-118. doi: http://doi.ieeecomputersociety.org/10.1109/PCEE.2002.1115214

Sadat-Mohtasham, H. (2011). *Transactional pointcuts for aspect-oriented programming.* (Doctoral dissertation), University of Alberta, Alberta, Canada.

Sato, Y., Inoguchi, Y., & Nakamura, T. (2011). On-the-fly detection of precise loop nests across procedures on a dynamic binary translation system. *Proceedings of the 8th ACM International Conference on Computing Frontiers*, 1-10. doi: 10.1145/2016604.2016634

Smith, L., & Bull, M. (2013). Java Grande Forum Benchmark Suite. from http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads/s2contents.html

Trifu, M., & Kuttruff, V. (2005). Capturing nontrivial concerns in object-oriented software. *Proceedings of the 12th Working Conference on Reverse Engineering*, 99-108. doi: http://dx.doi.org/10.1109/WCRE.2005.11

Tripp, O., Yorsh, G., Field, J., & Sagiv, M. (2011). HAWKEYE: effective discovery of dataflow impediments to parallelization. *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, 46*(10), 207-224. doi: 10.1145/2076021.2048085

Wu, P., Feautrier, P., Padua, D., & Sura, Z. (2002). Instance-wise points-to analysis for loop-based dependence testing. *Proceedings of the 16th International Conference on Supercomputing*, 262-273. doi: http://doi.acm.org/10.1145/514191.514228

Zhong, H., Mehrara, M., Lieberman, S., & Mahlke, S. (2008). *Uncovering hidden loop level parallelism in sequential applications*. Paper presented at the IEEE 14th International Symposium on High Performance Computer Architecture, Salt Lake City, UT.