ABSTRACT
In this paper, we describe the design and implementation of a parallelizable loop pointcut for an aspect-oriented compiler. Prior to this study, several prototype solutions existed for loop pointcuts, but the solutions were not very granular. In particular, they were 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, 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.

Identifying parallelizable loops is known to be a difficult problem, and as such, this study’s parallelizable loop pointcut implements a heuristic solution. Thus, the pointcut identifies many parallelizable loops as being parallelizable, but not all. For two test programs where the pointcut was unable to identify parallelizable loops, the inability to detect parallelizability was, surprisingly, somewhat beneficial. When those programs’ loops ran in parallel (as part of a non-aspect-oriented program), their calculated results were slightly different from the known theoretical results, but when run sequentially (with the aspect-oriented compiler), the calculated results matched the known theoretical results.

Categories and Subject Descriptors
D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement – Restructuring;
D.3.3 [Programming Languages]: Language Constructs and Features – Concurrent programming

General Terms
Performance, Design.

Keywords
parallelizable loops, aspect-oriented programming, loop pointcut.

1. INTRODUCTION
Loops sometimes contain operations that can be executed in a parallel manner via a multitasking environment or multiprocessor environment. When loops with parallelizable operations are executed in parallel, such parallelization can lead to improved performance. 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. 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 [1].

The primary goal of this study was to implement a parallelizable loop pointcut. To achieve that goal, we designed and implemented an algorithm that determines whether a given loop is parallelizable. 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 as part of an aspect-oriented compiler, 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.

Loops are more difficult to map to pointcuts than other constructs because loops don’t have standard arguments (like method arguments) or identifiers that can be used to identify them [2]. Determining the parallelizability of a loop is known to be a difficult problem, particularly when array references and nested loops are involved [3, 4] and can be NP-complete in certain circumstances [4, 5].

2. RELATED WORK
Harbulot and Gurd [2] implemented an aspect-oriented compiler, LoopsAJ, which included a pointcut for loops. Unlike this study’s resulting compiler, their compiler made no attempt to distinguish loops that are parallelizable. Their loop pointcut matched a particular type of loop – a loop with one exit node and one successor node. Figure 1 shows that type of loop, plus two other loop types. Harbulot and Gurd implemented just the first type of loop (one-exit-node, one-successor-node), because it has the most potential for aspectization functionality. Specifically, it can handle before, after, and around 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++)
This study’s aspect-oriented 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 [6]. Weaving loops at the bytecode level means that only one type of loop pointcut is needed rather than separate loop pointcuts for each type of Java source code loop - while, do, for, and for-each. On the other hand, if there were separate loop pointcuts for each type of Java source code loop, then several problems would exist: (1) the aspect code would be cluttered with the different loop pointcuts, and (2) the usage for each pointcut would be affected by the different styles exhibited by the source code programmers.

In [3], Blume et al. showed that if a loop exhibits cross-iteration dependence, then the loop is not parallelizable. There are three types of cross-iteration dependencies: (1) flow dependence, when a statement writes to a memory location first and then reads from that same memory location in a later iteration of the non-parallelized loop, (2) antidependence, when a statement reads from a memory location first and then writes to that same memory location in a later iteration of the non-parallelized loop, and (3) output dependence, when a statement writes to the same memory location in different iterations of the non-parallelized loop.

Most cross-iteration dependence analysis studies have focused on detecting parallelizability for loops that handle arrays [5, 7, 8]. The prominence of array-based loops in parallelizable loop studies is due to array-based loops’ penchant for handling numeric calculations, and such numeric calculations tend to benefit the most from parallelization. Bik and Gannon’s JAVAB tool [9] 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. 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 new drugs, and fluid dynamics for the design of aeropropulsion systems [5].

Sadat-Mohtasham [10] relied on a program dependence graph (PDG) as an underlying mechanism to implement an aspect-oriented loop pointcut as well as several other pointcuts. A PDG is a tool that models a program’s control dependencies, and in so doing, it provides hooks for developers to analyze different parts of a program (Ferrante, Ottenstein, and Warren, 1987). In compiling a target program with an aspect-oriented compiler, Sadat-Mohtasham’s tool created a PDG, used the PDG to identify constructs within the target program, and then used the identified constructs to weave advice into the resulting compiled target program. This study relied heavily on Sadat-Mohtasham’s research and, in particular, his work with PDGs.

When the abc aspect-oriented compiler was built, its primary goal was to be extensible [11]. Because of abc’s focus on extensibility, this study used abc as its starting-point aspect-oriented compiler. The abc compiler was designed and built with the help of Soot [12], a compiler tool which provides the analysis and code generation necessary for the matching and weaving operations inherent in abc’s aspect-oriented functionality. In implementing a parallelizable loop pointcut and weaving capabilities for the pointcut, this study added a significant amount of code to Soot’s framework.

3. METHODOLOGY

3.1 Parallelizable Loop Pointcut Syntax

In implementing a parallelizable loop pointcut, the first step was to define the pointcut syntax. This study implemented two loop pointcuts - loopPar and outerLoopPar. The loopPar pointcut should match all parallelizable loops – stand-alone loops, loops that surround other loops, and loops that are nested inside of other loops. The outerLoopPar pointcut should match only parallelizable loops that surround other loops. We implemented an outer loop pointcut and not an inner loop pointcut because parallelizing an outer loop tends to generate greater gains in efficiency [13].

The new parallelizable loop pointcuts were designed to work in conjunction with an args pointcut, which was used to identify a loop’s context. Here is the syntax for a loopPar pointcut with an associated args pointcut:

\[
\text{loopPar}() \&\& \text{args}((\text{min}, \text{max}, \text{step})
\]

3.2 Detect Loops with One Exit Node

This study implemented pointcuts for only one type of loop – a loop with one exit node and one successor node. For Java (source code and bytecode) loops, if there is one exit node, then there must be only one successor node. That’s because Java bytecode conditional statements are limited to one branching target. So in trying to look for loops with one exit node and one successor node, it is sufficient to simply look for loops with one exit node. To detect loops with one exit node, we used a PDG to identify all the instructions in a loop and then identify the number of instructions that are exit points from the loop.

3.3 Detect Whether a Loop is Parallelizable

The study’s next step was to design and implement an algorithm for detecting whether a given loop is parallelizable. The algorithm is explained in detail in Section 4.

In designing the algorithm, we necessarily took a conservative approach. If the algorithm indicates that a loop is parallelizable, then the loop is assured to be 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.

3.4 Weaving

The next step was to determine how to weave the new parallelizable loop pointcuts into a program’s loop join points. To implement the weaving process, this study relied 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). Jimple instructions are simpler than Java source code instructions,
thus making it easier for developers to analyze and manipulate a Java program’s operations [14]. Abc performs the weaving process on the resulting Jimple code. After the weaving takes place, abc converts the resulting Jimple code to bytecode.

To make abc’s built-in weaving tasks work for the new parallelizable loop pointcuts, we provided code that defines parallelizable loop pointcuts and parallelizable loop join points. In addition, we provided code to handle situations unique to loops. For example, after replacing targeted loops’ min, max, and step constant values with variables, the aspect-oriented weaver was able to weave max and step args values successfully, but it was unable to do so with min args values. 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. 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 so the starting point was before the initialization statement.

3.5 Create Test Programs
To measure the effectiveness of the study’s parallelizable loop pointcuts, multi-threaded test programs were needed. The test programs used the worker object creation pattern [15] to implement their multi-threaded functionality. The worker object creation pattern replaces 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. Treating the method call 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. Note how Figure 2’s program uses the worker object creation pattern. It instantiates a Runnable object so that Runnable’s run method can be called later on by a thread. The Runnable object r gets connected to a thread by passing r to a Thread constructor call.

```
void numThreads() {
    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() {
                proceedIfMin(t_min, t_max, t_step);
                }
            };
            threads[i] = new Thread(r);
        }
    }
    for (int i=0; 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 2. Loop parallelization advice using the proposed loopPar pointcut

Note that the code in Figure 2 contains around advice, so in an actual program, it would be embedded in an aspect file. Also note the around advice’s min, max, and step arguments. Those arguments are captured by the loopPar and args pointcuts.

Within the for loop, the min, max, and step arguments are used to assign interleaved values to t_min, t_max, and t_step. Finally, the call to proceed executes matching parallelizable loops while replacing their original min, max, and step values with the new t_min, t_max, and t_step values.

4. RESULTS
4.1 Algorithm for Detecting Loop Parallelizability
The following list provides an overview of the constraints checked for in the loop parallelizability detection algorithm:

1. The loop must be trivial.
2. No field modifications.
3. Limited uses and definitions.
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.

In their loop parallelization detection algorithm [9], Bik and Gannon described the constraints numbered 1, 2, 3, and 5 above. This study’s researchers originated the constraints numbered 4 and 6 above.

Constraint 1. The loop must be trivial. For the loop to be trivial, these characteristics must hold:

a) One exit node. This is necessary for implementing a loop pointcut that supports before, after, and around advice, and context exposure.
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 variable 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.

Constraints 1b, 1c, and 1d ensure that the loop index variable increments in a simple, consistent manner. Such simple, consistent incrementation makes it easier to assign subsets of the loops’ iterations to different threads.

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 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. More specifically, 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. (A block is a group of instructions that always execute in a sequence.)

or

ii. The definition is in a different block as the usage block, and within the same loop, and \( v' \in \text{Dom}(v) \), where \( v \) is the usage block, \( v' \) is the block that contains the definition, and \( \text{Dom}(v) \) is the set of blocks that dominates block \( v \). (A node \( x \) dominates node \( y \) if for every path from the starting node to node \( y \), the path goes through node \( x \).)

or

iii. The definition is outside the loop.

and

b) 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. In other words, all scalar variables defined in the loop must be dead upon exiting the loop.

**Constraint 3.** No field modifications, where 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. 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. Such points-to analysis can add significantly to the analysis’s complexity [8].

**Constraint 4.** Safe method calls only.

Do not allow “unsafe” method calls within the target loop. A method call is considered “unsafe” if:

- the called method contains a field modification(s),
- the called method contains a method call, or
- the method call contains an argument(s) that is a reference type.

The second criterion (the called method contains a method call) is considered unsafe because if there were method calls within the called method, 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.

The third criterion (the method call contains 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.

**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, then determine the lower bound index offset \( l \) (for \( \text{arr}[i + l] \)) and the upper bound index offset \( u \) (for \( \text{arr}[i + u] \)) 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 checking for array bases being aliases of each other and checking for array index expression variables being aliases of each other.

**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, it enables a different array element to be 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 could 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:

\[
\text{arr}[3] = i;
\]

If the \( \text{arr}[3] \) element were used after the loop and the loop’s iterations were executed in parallel, then the use’s value would be unpredictable.

### 4.2 Transcut Pointcut and Parallelizable Loop Pointcut Usage

To use this study’s parallelizable loop pointcuts, they need to be embedded within a transcut pointcut [10]. The transcut tool’s transcut pointcut groups other pointcuts together so that a group of join points can be matched by the transcut pointcut.

The aspect code in Figure 3 shows how to parallelize loops using the parallelizable loop pointcut. The bottom-half code and the subsequent “...” code are identical to the multi-threading aspect code shown in Figure 2, but the transcut code at the top 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.

```java
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;
                ...
            }
        } // end aspect LoopParTestAspect
}
```

**Figure 3.** Advice that uses a transcut pointcut and a loopPar pointcut to replace parallelizable loops with multi-threaded versions of those loops.
In Figure 3, inside the tc pointcut, a pointcut is defined in the traditional way - using the keyword pointcut. Outside of the transcut construct, we use the transcut by referring to the name tc, not loop (loop can be used only within the transcut construct). Note how the tc transcut is used within the around advice. The args pointcut retrieves the min, max, and step arguments from matched parallelizable loops.

### 4.3 Benchmark Program Results

This study tested its new parallelizable loop pointcut on four benchmark multi-threaded programs from the Java Grande Forum Benchmark Suite [16]. We compared output from the original multi-threaded programs with output from refactored versions of those programs, where the refactored programs achieved their parallelization by using this study's parallelizable-loop-pointcut aspect-oriented compiler. To compare outputs, print statements were added to the loop pointcut aspect code and to the loop threads in the non-aspect-oriented versions of the benchmark programs. Tables 1 and 2 show those output comparisons.

**Table 1.** Output for multi-threaded programs, aspect-oriented versions versus non-aspect-oriented versions

<table>
<thead>
<tr>
<th>Abbreviated trace output for aspect-oriented program</th>
<th>Trace output for non-aspect-oriented program</th>
</tr>
</thead>
<tbody>
<tr>
<td>coefficient 0.0</td>
<td>same</td>
</tr>
<tr>
<td>Computed value = 2.8729524964837996</td>
<td></td>
</tr>
<tr>
<td>Reference value = 2.8729524964837996</td>
<td></td>
</tr>
<tr>
<td>coefficient 1.0</td>
<td>same</td>
</tr>
<tr>
<td>Computed value = 0.0</td>
<td></td>
</tr>
<tr>
<td>Reference value = 0.0</td>
<td></td>
</tr>
<tr>
<td>coefficient 0.1</td>
<td>same</td>
</tr>
<tr>
<td>Computed value = 1.1161046676147888</td>
<td></td>
</tr>
<tr>
<td>Reference value = 1.1161046676147888</td>
<td></td>
</tr>
<tr>
<td>coefficient 1.1</td>
<td>same</td>
</tr>
<tr>
<td>Computed value = -1.8819691893398025</td>
<td></td>
</tr>
<tr>
<td>Reference value = -1.8819691893398025</td>
<td></td>
</tr>
<tr>
<td>coefficient 0.2</td>
<td>same</td>
</tr>
<tr>
<td>Computed value = 0.34429060398168704</td>
<td></td>
</tr>
<tr>
<td>Reference value = 0.34429060398168704</td>
<td></td>
</tr>
<tr>
<td>coefficient 1.2</td>
<td>same</td>
</tr>
<tr>
<td>Computed value = -1.1645642623320958</td>
<td></td>
</tr>
<tr>
<td>Reference value = -1.1645642623320958</td>
<td></td>
</tr>
<tr>
<td>coefficient 0.3</td>
<td>same</td>
</tr>
<tr>
<td>Computed value = 0.15238898702519288</td>
<td></td>
</tr>
<tr>
<td>Reference value = 0.15238898702519288</td>
<td></td>
</tr>
<tr>
<td>coefficient 1.3</td>
<td>same</td>
</tr>
<tr>
<td>Computed value = -0.8143461113044298</td>
<td></td>
</tr>
<tr>
<td>Reference value = -0.8143461113044298</td>
<td></td>
</tr>
</tbody>
</table>

**Table 2.** Output for multi-threaded programs, aspect-oriented versions versus non-aspect-oriented versions

<table>
<thead>
<tr>
<th>Abbreviated trace output for aspect-oriented program</th>
<th>Trace output for non-aspect-oriented program</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st original byte: 1</td>
<td>1st original byte: 0</td>
</tr>
<tr>
<td>1st decrypted byte: 0</td>
<td>1st decrypted byte: 0</td>
</tr>
<tr>
<td>1st original byte: 2</td>
<td>1st original byte: 2</td>
</tr>
<tr>
<td>1st decrypted byte: 2</td>
<td>1st decrypted byte: 2</td>
</tr>
<tr>
<td>1st original byte: 3</td>
<td>1st original byte: 3</td>
</tr>
<tr>
<td>1st decrypted byte: 3</td>
<td>1st decrypted byte: 3</td>
</tr>
</tbody>
</table>

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.

Table 1 shows the results for the first benchmark program, the Series program, which calculates Fourier coefficients. Table 2 shows the results for the next three benchmark program - Crypt, SOR, and SparseMatMult. The “Crypt” program performs encryption and decryption using the IDEA (International Data Encryption Algorithm) algorithm. The “SOR” program performs the successive over-relaxation (SOR) algorithm in solving a linear system of equations. The “SparseMatMult” program performs matrix multiplication by finding the product of all of the non-zero elements in an n x n sparse array.

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 x 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. 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 x 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. 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.

### 4.4 Software Repository

The software used in this study can be found at [http://captain.park.edu/jdean/nova/dissertation/parallelizableLoopAspectsSoftware.html](http://captain.park.edu/jdean/nova/dissertation/parallelizableLoopAspectsSoftware.html).

### 5. CONCLUSION

The aim of this work was 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 two goals were accomplished fully, with two loop pointcuts defined and implemented – loopPar, for matching all parallelizable loops, and outerLoopPar, for matching all parallelizable loops that are not inside another loop. The third goal was accomplished, but in order 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.

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. 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 in the study’s parallelizable loop detection algorithm and allow field modifications within target loops. But 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 [8]. Another way to make the algorithm more aggressive in terms of identifying parallelizable loops would be to incorporate Aho, et al.’s data dependence analysis algorithm [5], which is based on the work of [17]. 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.

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. Of course, the primary benefit of parallelizing loops is that it can lead to programs that execute faster.

6. REFERENCES