PROGRAMMER-ASSISTED AUTOMATIC PARALLELIZATION

by

Diego Huang

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2011 by Diego Huang
Abstract

Programmer-Assisted Automatic Parallelization

Diego Huang

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2011

Parallel software is now required to exploit the abundance of threads and processors in modern multicore computers. Unfortunately, manual parallelization is too time-consuming and error-prone for all but the most advanced programmers. While automatic parallelization promises threaded software with little programmer effort, current auto-parallelizers are easily thwarted by pointers and other forms of ambiguity in the code. In this dissertation we profile the loops in SPEC CPU2006, categorize the loops in terms of available parallelism, and focus on promising loops that are not parallelized by IBM’s XL C/C++ V10 auto-parallelizer. For those loops we propose methods of improved interaction between the programmer and compiler that can facilitate their parallelization. In particular, we (i) suggest methods for the compiler to better identify to the programmer the parallelization-blockers; (ii) suggest methods for the programmer to provide guarantees to the compiler that overcome these parallelization-blockers; and (iii) evaluate the resulting impact on performance.
Dedication

This is dedicated to Dad and Mom, who have made countless sacrifices to allow me to pursue my dreams.
Acknowledgements

I would like to thank my supervisor Professor Greg Steffan for his support, patience, and guidance.

I would also like to thank Yaoqing Gao and Kit Barton from IBM Canada for their help with the IBM XL C/C++ compiler, and Peng Wu from IBM US for her help with the DProf dependence profiling tool.

To my fellow graduate students in EA305, EA306, LP392 and LP492, thank you for always being there for discussions and ideas.
# Contents

1 Introduction 1
   1.1 Improving Support for Programmer Guarantees 2
   1.2 Research Goals 3
   1.3 Thesis Organization 3

2 Background 5
   2.1 Run-time Checks 6
   2.2 Speculative/Optimistic Parallelization 7
   2.3 Programmer Guarantees 8

3 Analyzing Loops in SPEC CPU2006 11
   3.1 Loop Coverage Profiling 11
   3.2 Dependence Profiling 12
   3.3 Analyzable loops 15
   3.4 Loops with Parallel Instances 18
   3.5 Loops with Few Dependences 20
   3.6 Loops with Low Dependence Frequencies 22
   3.7 Loops with Large Average Independence Window Sizes 24
   3.8 Loops of Interest 24
   3.9 Summary 27

4 Programmer Guarantees 28
A Loops of Interest in SPEC CPU2006

A.1 Loops of Interest for Manual Parallelization ........................................... 60
A.2 Loops of Interest for Run-time Checks ..................................................... 60
A.3 Loops of Interest for Speculative Parallelization ...................................... 60
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>SPEC CPU2006 loops analyzable using DProf</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>Parallel instances of SPEC CPU2006 loops</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>Number of dependences of SPEC CPU2006 loops</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>Dependence frequencies of SPEC CPU2006 loops</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>Independence window sizes of SPEC CPU2006 loops</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>SPEC CPU2006 benchmarks that speed up with programmer guarantees</td>
<td>48</td>
</tr>
<tr>
<td>5.2</td>
<td>SPEC CPU2006 benchmarks that slow down or don’t speed up with programmer</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>guarantees</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The advent of multithreaded-multicore processors like IBM’s Power line of processors has made it clear that software must be parallel to fully benefit from the potential performance of these machines. Parallelizing software manually is a difficult, time-consuming, and error-prone process: the programmer must worry about correctness, synchronization of shared data, and deadlocks, while navigating trade-offs such as communication versus computation and load-balancing.

Instead we would prefer the compiler to parallelize software automatically, to handle these complex decisions and trade-offs for the programmer. This has motivated efforts like IBM’s XL C/C++ auto-parallelization facility that is increasingly capable of parallelizing many complex code patterns. However, there remain many sources of ambiguity in most programs that can block the efforts of parallelizing compilers; the most prominent examples are the use of pointers and complex and heap-based data structures, but ambiguity also results from the use of complex array indexing, function pointers, and forms of complex control flow.

There are three main ways for a compiler to parallelize in the presence of such ambiguity:

1. **Guards and run-time checks**: if the ambiguity is localized and can be quickly disambiguated at run-time (e.g., by comparing two pointers), then the compiler can generate code to resolve this condition and predicate the parallel version of the code
on the result of this check [26, 28].

2. **Optimistic parallelization**: when there are numerous ambiguous memory references, systems for optimistic parallelization such as Thread-Level Speculation (TLS) [14, 29] and Transactional Memory (TM) [12, 13] can be used to detect and recover from any that conflict at run-time; however such systems are not yet commonly supported in hardware, and have nearly prohibitive overhead when supported solely in software.

3. **Programmer guarantees**: the programmer resolves the ambiguity by providing an appropriate guarantee to the compiler (e.g., by guaranteeing the independence of two pointers via the `restrict` keyword).

### 1.1 Improving Support for Programmer Guarantees

In this dissertation we focus on the third option above: programmer guarantees to better-enable auto-parallelizing compilers. In particular, we focus on the case where a programmer hopes to parallelize a legacy application with minimal effort—as opposed to the possibility of rewriting the application using a parallel language or library. In this case the goal is to allow the programmer to best leverage the abilities of the parallelizing compiler, by performing only the tasks that the compiler is unable to.

The advantages of pursuing programmer guarantees are: (i) there is no run-time overhead, since ambiguities are resolved statically at compile-time; (ii) code remains readable, unlike explicit parallelization where code must often be significantly restructured and expanded for communication and synchronization; (iii) code remains portable, since we avoid any machine-specific code modifications and defer that responsibility to the compiler; (iv) most guarantees can easily be exploited by existing compiler transformations with little modification, as the guarantees are only a modification of previously-collected dependence information; and (v) an effective guarantee suggests future work to replace it with an automated guarding process, or
the potential for applying optimistic parallelization (should efficient support for that become available).

While there are many advantages, to be fair there are also disadvantages to programmer guarantees: they are not automatic and require manual effort from the programmer, and they are limited to the current guarantee/pragmas interface and parallelization abilities of the compiler. Beyond these, the bigger problem (and hence opportunity for improvement) for programmer guarantees is that they are not clearly encouraged by the feedback from current parallelizing compilers such as XL C/C++, because they do not give complete/detailed information on why parallelization failed for a given loop (e.g., variable names), nor guide the programmer to promising loops that are "close" to being successfully parallelized.

1.2 Research Goals

This thesis focuses on improving the support for programmer guarantees in automatic parallelizing compilers. We accomplish this through the following goals:

1. To analyze the characteristics of the dependences in hot loops not automatically parallelized by XL C/C++ and find loops that are profiled to be parallel at run-time.

2. To explore the programmer guarantees required to allow XL C/C++ to parallelize the loops.

3. To suggest better compiler feedback to the programmer for easier usage of programmer guarantees.

1.3 Thesis Organization

This dissertation is organized as follows: Chapter 2 provides an overview of the approaches that auto-parallelizing compilers can use to resolve source code ambiguities, Chapter 3 analyzes the
loops in SPEC CPU2006 to find run-time parallel loops, Chapter 4 investigates techniques on improving the support for programmer guarantees to parallelize the run-time parallel loops found in the previous chapter, Chapter 5 compares the program speedup of the benchmarks auto-parallelized with programmer guarantees with the original benchmarks auto-parallelized without guarantees, and Chapter 6 summarizes our contributions and discusses potential future research directions.
Chapter 2

Background

Automatic parallelization off-loads the work of parallelization from the programmer to the compiler. The compiler is responsible for (i) analyzing and identifying parallel regions in the program, (ii) determining if the parallel regions will likely speed up when parallelized, and (iii) generating the parallel code. The step to analyze and identify parallel regions is the most difficult one because the compiler needs to prove that memory accesses across different threads of execution do not interfere with each other. Since most of the execution time in programs is spent in loops, most research is in automatically parallelizing loops [5,19,30,35,36]. However, ambiguities in the loops can confuse the compiler and prevent it from parallelizing the loops. For example, Listing 2.1 shows a loop that assigns the value at address \( p \) to the array \( a[] \). If \( p \) points to an element in \( a[] \), then the loop will have loop-carried dependences and cannot be parallelized. If \( p \) does not point to an element of \( a[] \), then the loop can be parallelized. Even with the most recent pointer analyses [7, 15, 16, 21], the compiler still cannot disambiguate pointers that are, for example, input-dependent.

In this chapter, we review the three main methods that compilers use to deal with code ambiguities: run-time checks, speculation and the most closely-related work on programmer guarantees.
2.1 Run-time Checks

Run-time checks, also known as guards and dynamic memory disambiguation, are short pieces of code that the compiler generates to resolve an ambiguity at run-time. Depending on the outcome of the check, a sequential or parallel version of the original loop is executed. For example, Listing 2.2 shows the loop from Listing 2.1 with a run-time check (the if-statement) that checks if the pointer \( p \) points to anywhere within the array \( a[] \). If \( p \) points to anywhere within the array \( a[] \), then there are loop-carried dependences and the sequential version of the loop is executed. If \( p \) points outside of the array \( a[] \), then the parallel version of the loop can be executed.

Run-time checks can be as simple as single if-statements that can be executed in \( O(1) \) time [25, 26], or can be as complicated as checks involving loops and sorting in \( O(n \log n) \).
Run-time checks are limited to cases where it is possible to disambiguate memory references using a simple expression. They are not suitable for cases where the check may have large overheads, such as array indirections of the form \( a[b[i]] \) which may need a \( O(n^2) \) check, and for cases where a function’s side-effects need to be checked.

### 2.2 Speculative/Optimistic Parallelization

Speculative parallelization techniques, like run-time checks, delay the task of dependence checking until run-time. The compiler parallelizes the program optimistically, even in the presence of ambiguous memory references, and relies on the hardware or run-time software library to keep track of the memory references and check for dependences. If a dependence is detected at run-time, the speculative thread needs to be squashed, its program changes undone, and re-executed either in parallel or sequential.

There are two flavours of speculative parallelization: Transactional Memory (TM) \([12, 13, 24, 27]\) and Thread-Level Speculation (TLS) \([14, 23, 29, 38]\). The main difference between the two is that TLS guarantees commit order, whereas TM does not. As a result, TLS is usually applied to loops, while TM to parallel regions.

There are also software and hardware flavours of speculative parallelization. While software systems can be deployed on existing hardware, the overheads are prohibitively high and it’s very difficult to gain performance improvements. Hardware systems have lower
overheads, but none are in production yet, so the research in this area has mostly relied on simulation.

In the near future, hybrid transactional systems [9, 11] will likely be released first. These systems have hardware that supports small speculative threads. When the number of speculative memory accesses grow and overflow the hardware buffers, the speculative thread is moved to software.

2.3 Programmer Guarantees

The third method for resolving source code ambiguities is to allow the programmer to assist the compiler’s analyses by specifying a guarantee about their code. The guarantee allows the compiler to parallelize without the run-time overheads of run-time checks and speculative techniques, but requires manual effort by the programmer to verify the guarantee, which is why it is important that this manual effort be minimized.

In the running example from Listing 2.1, the compiler cannot prove that accesses to \( a[\] \) will not alias with \( *p \). However, the programmer can provide a guarantee to the compiler that accesses to \( a[\] \) and \( *p \) will not alias, through a \#pragma disjoint as shown in Listing 2.4. The guarantee allows the compiler to perform more aggressive optimizations, such as loop-invariant code motion (moving the load of \( p \) outside the loop) or auto-parallelization.

A well-known programmer guarantee is C99’s restrict keyword [1], which allows the programmer to guarantee that the memory pointed to by a pointer will only be accessed through that pointer only. Though it has been shown that using the restrict keyword alone
does not result in significant performance improvements in SPEC CPU2000 [25] with regards to auto-vectorization and instruction-level parallelism, we use the more flexible \#pragma disjoint provided by IBM’s XL C/C++ compiler, which indicates that only the pointers specified in the pragma are independent, and does not imply anything of the other pointers in the program. Koes et. al. propose a similar \#pragma independent [18] and they evaluate its usefulness with traditional optimizations (\texttt{gcc -O2}), whereas we focus on its benefits for thread parallelization.

OpenMP [3] essentially provides a collection of programmer guarantees to the compiler that certain loops are parallel—however the onus is on the programmer to identify and manage any dependences. This is similar to Fortran 2008’s \texttt{DO CONCURRENT} loop [2], and Galois’ unordered \texttt{for each} loop [20]. In contrast, in this work we pursue the idea that the programmer should only provide guarantees related to specific ambiguities that are limiting the auto-parallelization efforts of the compiler, such as pointer aliases or function side-effects.

There has been previous work in proposing new programmer guarantees for pipeline parallelism [6, 31, 33]. Our work on programmer guarantees focuses instead on \texttt{DOALL} parallelism, because this is the type of loops that XL C/C++ parallelizes automatically.

Interactive parallelization tools guide programmers as to where to place the guarantees in their code. These help in minimizing the amount of effort required by the programmer to specify the guarantees. Previously proposed tools [8, 22] suggest to the programmer where to place OpenMP pragmas. However, this requires that the programmer verify that \texttt{nothing} in the loop may cause loop-carried dependences. In our work, we suggest programmer guarantees specific to what is blocking auto-parallelization (such as ambiguous pointers), which is less work for the programmer to verify because ambiguities that the compiler could resolve automatically are filtered out. Koes et al. [18] propose a tool which suggests to the programmer where to place pragmas to disambiguate pointers. However, their work focuses on traditional compiler optimizations, and not auto-parallelization. There has been previous work in using dependence profiling to suggest to the programmer possible guarantees that
would lead to parallelization, such as Prospector [17], Paralax [33], and the tools proposed by Thies et al. [31] and Tournavitis et al. [32]. These tools target pipeline parallelism, whereas we focus on DOALL parallelism. In the next chapter we describe the process that we propose the compiler use to find suitable DOALL loops to suggest programmer guarantees.
Chapter 3

Analyzing Loops in SPEC CPU2006

The automatic parallelization facility in IBM’s XL C/C++ compiler allows the programmer to parallelize an application with minimal effort. While many loops can be automatically parallelized, there are also many that contain ambiguous pointers and function calls that hinder parallelization.

In this chapter we provide an in-depth analysis of the characteristics of the loops that were not parallelized by XL C/C++ V10.1 in the C/C++ benchmarks of the SPEC CPU2006 benchmark suite. This industry-standard benchmark suite has many loops and many types of loops which XL C/C++ cannot parallelize, and hence is a good subject for the exploration of increasing parallelization opportunities via programmer guarantees.

We analyze the loops by first profiling their run-time coverage. For the loops that cover 10% or more of the run-time, we instrument them for dependence profiling. In the next subsections, we discuss the loop profiling tool, the dependence profiling tool, and the results from dependence profiling.

3.1 Loop Coverage Profiling

We perform loop profiling to find hot loops of 10% or more of run-time coverage. A simple loop profiler was built based on the DProf [10,37] dependence profiler infrastructure, discussed
Listing 3.1: Sample loop profiling instrumentation

in the next section. The entry point of the loop is automatically instrumented by a compiler pass with a function call to `profile_region_begin()`, which starts a timer, and the exit points are instrumented with a function call to `profile_region_end()`, which stops the timer. The loop back-edge is instrumented with `profile_event()` to count the number of iterations. Listing 3.1 shows an example of an instrumented loop ready for loop profiling. Next section offers more details about the DProf profiler that this is based on.

### 3.2 Dependence Profiling

DProf is a research tool developed by IBM to profile for run-time dependences. It consists of a compiler pass inside the XL C/C++ compiler for instrumenting loops, and a software run-time library for analyzing the run-time memory accesses.

Listing 3.2 shows an example of a loop after DProf instrumentation. The compiler pass instruments the loop entry with a function call to `profile_region_begin()`, the loop exits with a call to `profile_region_end()`, the loop back-edges with `profile_event()`, and the loop’s memory accesses with `profile_access()`. Listing 3.2 shows an example of a loop instrumented by DProf. The argument for
Listing 3.2: Sample loop dependence profiling instrumentation

__profile_region_begin(0);
for (i = 0; i < 1000; i++) {
  /* memory write access id 5 */
  __profile_access(&a[i], 5);

  /* memory read access id 7 */
  a[i] += __profile_access(&b[i], 7);

  /* loop id 0, event id 0 (loop back edge) */
  __profile_event(0, 0);
}
/* loop id 0 */
__profile_region_end(0);

__profile_region_begin(), __profile_region_end(), and the first argument of __profile_event() is the global unique ID of the loop, which is an integer composed of bits representing the file ID, procedure ID, and a procedure-local loop ID. The second argument of __profile_event() is the event ID, which is always 0 for loop back-edges. The function __profile_access() takes as arguments the address of the memory accessed, and the memory access ID, which is composed of bits representing the file ID, procedure ID, and procedure-local memory access ID. DProf generates information files that the run-time library uses, containing detailed information for each of the IDs, such as, file name for each file ID, procedure name for each procedure ID, loop line number for each loop ID, and whether a memory access is a read or write for each memory access ID.

Functions called from within the loop which are in the same compilation unit are recursively instrumented, while called functions not in the same compilation unit cause DProf to generate a message that asks the user to compile the other compilation unit with a compiler option that
tells the compiler to instrument that function. This means that the user has to compile with DProf several times to make sure that the entire function call chain from the loop is properly instrumented. Fortunately, the user can automate this process with a script by invoking DProf in a loop until no more messages are generated. This cumbersome process is because DProf is currently implemented as a compile-time pass, and can be eliminated if DProf instrumentation is moved to link-time when the whole program is available.

DProf has limitations: it cannot instrument loops that contain function pointers or bit-fields because they cannot be accurately profiled for dependences; it will not instrument loops with multiple entry points, and has limited support for loops with multiple exit points; and it uses a large amount of memory, which prevents it from profiling loops with extremely large loop bodies. As with any run-time profiling, the results from DProf are dependent on the input files used, so loops that are found to be free of dependences at run-time for a given input still need to be verified manually by the programmer. Due to these limitations, we first examine the amount of loops that DProf can instrument and analyze in the next section, and then we show the results obtained from DProf: (i) the percentage of loop instances that are parallel, (ii) the number of dependences per loop, (iii) the frequency of dependences and (iv) the independence window size. The definitions of each of these measurements are defined in the following sections. The frequencies of dependences and the independence window sizes do not help find appropriate run-time parallel loops for programmer guarantees; however, these metrics are useful for speculative parallelization, and so we also report them here.

We have incorporated several improvements to DProf to allow for easier profiling of SPEC CPU2006:

1. the ID numbers were changed from 32-bit to 64-bit integers to allow instrumentation of large loops;

2. a command-line option was added to allow users to specify which loops to profile, instead of having to mark the loops in the source code, to allow loops to be instrumented
3. Analyzing loops in SPEC CPU2006

3. Analyzable loops

Figure 3.1 shows the cumulative percent run-time of the hot loops that were (i) not auto-parallelized and instrumented by DProf (analyzable), (ii) not auto-parallelized and not instrumented by DProf (unknown), and (iii) auto-parallelized loops (parallelized). Cumulative percent run-time is the sum of the run-times of the loops in that category, regardless of loop nesting. In other words, nested loops can be multiply-counted, and the cumulative percent run-time can be over 100%. We use this measurement because it gives an indication of the
Figure 3.1: Cumulative run-time of hot loops broken down into loops that are already parallelized by XL C/C++ (parallelized), loops instrumented by DProf (analyzable), and loops that could not be instrumented with DProf (unknown).

The importance of the loop in the run-time execution. Table 3.1 shows the number of loops in each of these categories.

DProf, being a research tool, is able to analyze a significant fraction of the loops in SPEC CPU2006, especially in the benchmarks ASTAR, DEALII, H264REF, HMNER, LBM, LIBQUANTUM, MILC, NAMD, POVRAY and SPHINX3. Six of these benchmarks are from the floating point benchmarks, while only 4 are from the integer benchmarks. DProf works best with loops found in floating-point programs because their loops tend to be countable loops that have a single entry, a single exit and a counter that increases by a fixed amount, which are easier to handle by DProf.

The benchmarks that DProf has trouble analyzing are the integer benchmarks of SPEC CPU2006. These benchmarks tend to have loops with multiple exits, multiple back-edges,
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Parallelized</th>
<th>Unknown</th>
<th>Analyzable</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>astar</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>bzip2</td>
<td>0</td>
<td>7</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>dealII</td>
<td>0</td>
<td>5</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>gcc</td>
<td>0</td>
<td>7</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>gobmk</td>
<td>0</td>
<td>18</td>
<td>12</td>
<td>30</td>
</tr>
<tr>
<td>h264ref</td>
<td>0</td>
<td>9</td>
<td>15</td>
<td>24</td>
</tr>
<tr>
<td>hmmer</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>lbm</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>libquantum</td>
<td>2</td>
<td>0</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>mcf</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>milc</td>
<td>0</td>
<td>1</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>namd</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>omnetpp</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>perlbench</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>povray</td>
<td>0</td>
<td>7</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>sjeng</td>
<td>2</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>soplex</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>sphinx3</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>0</td>
<td>13</td>
<td>2</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 3.1: Number of loops that (i) were parallelized by XL C/C++ V10.1, (ii) failed to profile with DProf, and (iii) were successfully profiled with DProf.

and are not countable. BZIP2, GCC, MCF, OMNETPP, PERLBENCH, SJENG, SOPLEX and XALANCBMK have loops with goto statements, multiple loop exits, function returns, or C++ exceptions. PERLBENCH and XALANCBMK have loops with function pointers or virtual functions, and GCC has loops that access bit-fields; both of these types of loops cannot be instrumented by DProf for accurate dependence profiling. GOBMK fails to finish executing
Figure 3.2: Cumulative run-time of hot loops broken down into loops that are always parallel (100%), loops that have most instances parallel (76-99% and 51-75%), and loops that have few parallel instances (26-50% and 0-25%).

because it is a recursive program and the recursion causes the hot loops to use up more than the 64GB limit of memory.

Despite DProf not being able to instrument all the loops in the C/C++ benchmarks of SPEC CPU2006, it is able to instrument and analyze a significant fraction of the loops. In the next sections we analyze the DProf results with regards to: (i) loops with parallel instances, (ii) loops with few dependences, (iii) loops with low dependence frequencies, and (iv) loops with large average independence window sizes.

### 3.4 Loops with Parallel Instances

The percentage of loop instances (invocations) which are run-time parallel helps to determine the type of parallelization that should be applied to the loop. Figure 3.2 breaks down the
### Table 3.2: Number of loops in each of the run-time parallel categories in Figure 3.2.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>0-25%</th>
<th>26-50%</th>
<th>51-75%</th>
<th>76-99%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>astar</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>bzip2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>dealII</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>gcc</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>gobmk</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>h264ref</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>hmmer</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lbm</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>libquantum</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mcf</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>milc</td>
<td>13</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>namd</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>omnetpp</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>perlbench</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>povray</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sjeng</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>soplex</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sphinx3</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

The loops with ‘100%’ parallel instances are the loops that never encountered run-time dependences in any loop instance, and so could potentially be parallel loops. These run-time parallel loops can be parallelized by the programmer explicitly parallelizing the loop, by the programmer specifying a guarantee to resolve the ambiguity that’s causing the compiler not to
parallelize, or by the compiler generating a run-time check to resolve the ambiguity at run-time. The benchmarks DEALII, LBM, MILC and XALANCBMK have a significant fraction of loops of this type: 166%, 99%, 54%, and 25% in cumulative percent run-time respectively. These loops are ideal for exploring increasing parallelization opportunities with programmer guarantees, and in chapter 4 we investigate how the run-time parallel loops in these benchmarks can be parallelized with programmer guarantees.

The loops with ‘76-99%’ and ‘51-75%’ of run-time instances are not always parallel, and so they cannot be parallelized with programmer guarantees; however, these loops may be worth parallelizing with run-time checks because a large fraction of the instances are parallel. The benchmarks ASTAR, DEALII, GOBMK and SJENG have loops of this type.

The rest of the loops, ‘26-50%’, and ‘0-25%’ are likely not worth parallelizing because they do not have enough parallel instances to likely improve performance.

### 3.5 Loops with Few Dependences

The number of unique loop-carried dependence pairs in a loop gives an indication of the difficulty required to parallelize the loop. If a loop does not have dependences, it is potentially parallelizable using programmer guarantees. If a loop does have dependences, loops with fewer dependences are generally easier to parallelize by synchronizing the few shared data. DProf profiles for run-time true loop-carried dependences, maps them back to source code variables, and reports the pairs of variables that are causing the dependences. DProf reports pairs of variables only once, even if they cause dependences with multiple run-time memory accesses (e.g. array references). Reduction and privatizable variables detected by the auto-parallelizer are not instrumented by DProf, and hence are not reported as dependences. Figure 3.3 breaks down the cumulative percent run-time of analyzable loops according to the number of true dependences detected at run-time.

The bars with ‘0’ dependences are the run-time parallel loops that this dissertation
Figure 3.3: Cumulative run-time of hot loops broken down into loops with no true dependences (0), loops with few dependences (1-2, 3-5) and loops with many dependences (>5).

focuses on. These loops with no dependences are also the same loops found with 100% parallel instances (section 3.4) and would later be parallelized with programmer guarantees in chapter 4.

The bars with ‘1-2’ and ‘3-5’ dependences are the loops that have few dependences that the programmer could possibly eliminate by performing some loop transformations such as loop splitting or data synchronization. The benchmarks ASTAR, DEALII, HMmer, LIBQUANTUM, NAMD, SOPLEX and SPHINX3 have loops of this type with few dependences. The loops with ‘>5’ dependences are likely not worth parallelizing because they require more effort from the programmer to parallelize correctly. Several of these loops contain more than 100 dependences.
minimized if loop dependences do not occur very frequently. DProf measures the frequency of
frequencies detected at run-time.

3.6 Loops with Low Dependence Frequencies

The metrics discussed in these next two sections do not help find appropriate run-time parallel
loops for applying programmer guarantees; however, they are of interest for speculative
parallelization, and so we provide a brief discussion here.

Speculative parallelization techniques such as TM and TLS work best with loops that have
infrequent dependences. This is because the overhead of rolling back speculative threads is
minimized if loop dependences do not occur very frequently. DProf measures the frequency of
each dependence over the number of iterations and we average all the dependence frequencies
of one loop and report it as the loop’s average dependence frequency. Figure 3.4 breaks down
the cumulative percent run-time of analyzable loops according to their average dependence
frequencies detected at run-time.

Figure 3.4: Cumulative run-time of hot loops broken down into loops with no dependences
(parallel), loops with low (1-10%), medium (11-20%), and high (21-99%) average dependence
frequencies, and sequential loops (100%).
Figure 3.5: Cumulative run-time of hot loops broken down into loops with an average independence window size as large as the number of iterations (parallel), loops with large (≥ 9), medium (5-8) and small (2-4) independence window sizes, and sequential loops (1).

The loops labelled ‘parallel’ are the loops that have ‘0%’ dependence frequency and hence are run-time parallel. The loops with ‘1-10%’ dependence frequencies are the loops with low enough dependence frequencies that would be suitable for software TM or TLS. The benchmarks ASTAR, BZIP2, GOBMK and POVRAY contain loops of this type. With hardware support for TM and TLS, the lower overheads will support parallelization of loops with higher dependence frequencies, such as the ‘11-20%’ loops in MILC. The loops with ‘21-99%’ and ‘100%’ average dependence frequencies will likely cause too many aborts and roll-backs and hence they are likely not worth parallelizing with speculative methods.
3.7 Loops with Large Average Independence Window Sizes

The average independence window size of each loop is the average consecutive number of iterations without loop-carried dependences. In other words, it is the average number of loop iterations that can execute in parallel without dependences. This measurement is useful for speculative parallelization because it gives an estimate of the number of threads that should be spawned to minimize the number of conflicts and roll-backs caused by dependences. Figure 3.5 breaks down the cumulative percent run-time based on the average independence window sizes of the profiled loops.

The ‘parallel’ bars show the run-time parallel loops, which have independence window sizes as large as the iteration count. The loops with ‘2-4’ independence window sizes work best with 2-4 threads, such as for ASTAR, MILC and PERLBENCH, and so these are suitable for the current generation of CMPs that have 2 to 4 cores. The loops with ‘5-8’ and ‘≥ 9’ independence window sizes are suitable for the next generation of CMPs with many cores, such as BZIP2, GOBMK and SJENG. Loops with window size of ‘1’ are sequential (non-parallel) loops.

3.8 Loops of Interest

The previous 4 sections found loops that were of interest to programmers and compiler developers working on parallelization in the presence of ambiguities. Here we summarize the loop characteristics that are most suitable for manual parallelization, run-time checks, speculative parallelization and programmer guarantees, and refer the reader to the list of loops found for each parallelization option.

**Manual Parallelization** The programmer interested in parallelizing a program manually needs to focus on loops with a high percentage of parallel instances to maximize speedup, a low number of dependences to minimize the parallelization effort, and a low frequency of
dependences to maximize speedup. These are listed in Table A.1 in Appendix A.

**Run-time checks** The compiler developer looking into improving support for run-time checks needs to focus on loops with a high percentage of parallel instances to maximize speedup, and a low number of dependences to minimize the overhead of the check. These are listed in Table A.2 in Appendix A.

**Speculative Parallelization** For compiler developers working on speculative parallelization, loops of interest have a high percentage of parallel instances to maximize speedup, a low number of dependences and a low frequency of occurring dependences to minimize conflicts and roll-backs, and a high average independence window size to maximize the number of threads. These are listed in Table A.3 in Appendix A.

**Programmer Guarantees** In our work to improve the support of programmer guarantees,
Table 3.4: Other interesting loops not parallelized by XL C/C++. These were not found with DProf because the HMMER loop does have dependences (not a run-time parallel loop) and the GCC loop is not a hot loop.

we focus on the loops with 100% of parallel instances. In other words, the loops with 0 dependences, 0% dependence frequency, and a minimum independence window size as large as the number of iterations. Table 3.3 lists the run-time parallel hot loops that with more than 2 iterations that XL C/C++ V10.1 could not automatically parallelize, their characteristics, and the reason that XL C/C++ failed to parallelize them. Some of the run-time parallel loops found by DProf only had 1 iteration, which are not interesting, so they are not included in the table. Two loops in MILC and two loops in DEALII only had 3 iterations each, and XL C/C++ chooses to unroll these instead of parallelizing them. For the other loops, they were not automatically parallelized because of potential loop-carried dependences due to pointer aliasing/array overlap or function call with side effects. We also found other interesting loops for applying programmer guarantees that were found through manual investigation of the hot functions in SPEC CPU2006 [34]. These loops, listed in Table 3.4, were not found with DProf because the HMMER loop does have dependences (not a run-time parallel loop) and the GCC loop is not a hot loop. The HMMER loop is not automatically parallelized because of potential pointer aliasing causing loop-carried dependences, and the GCC loop is not parallelized because of indirect array references (i.e. array subscript is another array subscript). Even though the GCC loop will likely not be worth parallelizing due to its low run-time coverage, we include this loop in our analysis because it has a different type of parallelization-blocking ambiguity not found in the other loops. In the next chapter we parallelize the loops from both these tables.

<table>
<thead>
<tr>
<th>Loop</th>
<th>Benchmark</th>
<th>Run-time</th>
<th>Invocations</th>
<th>Iterations</th>
<th>Reason Not Parallelized</th>
</tr>
</thead>
<tbody>
<tr>
<td>fast_algorithms.c:133</td>
<td>hmmmer</td>
<td>72.1%</td>
<td>16115043.5</td>
<td>127.92</td>
<td>pointer aliasing</td>
</tr>
<tr>
<td>cselib.c:231</td>
<td>gcc</td>
<td>0.01%</td>
<td>94028.11</td>
<td>9.73</td>
<td>array indirection</td>
</tr>
</tbody>
</table>
with programmer guarantees.

3.9 **Summary**

Even though parallelism in SPEC CPU2006 is limited, there are still some interesting opportunities in parallelizing the loops that were detected to be parallel at run-time with programmer guarantees. In particular, the benchmarks DEALII, LBM, MILC, XALANCBMK, HMMER and GCC have interesting loops and we describe in the next chapter how these loops can be parallelized with programmer guarantees and how the compiler can ease the job of the programmer.
Chapter 4

Programmer Guarantees

In this chapter, we investigate how programmer guarantees can be improved to parallelize loops that a compiler failed to automatically parallelize; in particular, we focus on the most interesting loops found in the previous chapter (listed in Tables 3.3 and 3.4). We first discuss the reasons that make IBM’s XL C/C++ compiler a good platform for exploring the improvement of programmer guarantee support, and then we describe how each of the interesting loops found in the tables are parallelized with programmer guarantees.

4.1 Programmer Guarantee Support in XL C/C++

To allow the compiler to suggest possible guarantees to the programmer, and to minimize the effort required by the programmer to verify those suggestions, the compiler needs to filter out non-hot loops that would not lead to program speedups, and loops that are provably not parallel. To do this, we suggest that the compiler: (i) find hot loops by profiling the run-time coverage of the loops that it could not auto-parallelize; (ii) find run-time parallel loops by profiling the previously-found hot loops for run-time loop-carried dependences; (iii) report in detail the reasons that each run-time parallel hot loop was not automatically parallelized and suggest possible programmer guarantees that would lead to successful parallelization.

IBM’s XL C/C++ compiler is ideal for investigating improvements in programmer
guarantees because it has (i) a mature automatic parallelizer, available for at least the
last 6 major releases of the compiler, (ii) facilities for profile-directed feedback (PDF) to
guide compiler optimizations, such as parallelization, (iii) support for dependence profiling
through DProf, (iv) parallelization reports that inform the programmer which loops have been
successfully parallelized and the reason for the loops that it failed to parallelize, and (v) some
existing programmer guarantees that assist the analysis passes of the compiler.

XL C/C++ provides facilities for profile-directed feedback (PDF) that use run-time
profiling information to guide optimizations based on metrics such as function call counters,
basic block counters, and cache misses. We propose that this same framework be extended, for
when automatic parallelization is attempted, to also perform loop profiling and dependence
profiling. Basic block counters can be used to find hot loops, or loops that are at least a
significant fraction of execution time. This information can in turn be used to reduce the set of
loops for which dependence profiling is performed. The results of dependence profiling can be
used to further restrict the set of loops for which guarantees are proposed to the programmer,
to those loops that are at least run-time parallel according to the profile information.

XL C/C++ provides feedback to the programmer regarding parallelization-blockers via the
-qreport option. Listing 4.1 shows an example of a parallelization report generated by
XL C/C++ V10.1 on a program that has a loop at line 14 which is parallelized, and a loop
at line 20 which could not be parallelized due to ambiguous pointers. The first column in the

<table>
<thead>
<tr>
<th>Source</th>
<th>Source</th>
<th>Loop Id</th>
<th>Action / Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Line</td>
<td>--------</td>
<td>----------------------</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>-------</td>
<td>---------------------</td>
</tr>
<tr>
<td>0</td>
<td>14</td>
<td>1</td>
<td>Loop has been automatically parallelized.</td>
</tr>
<tr>
<td>0</td>
<td>20</td>
<td>2</td>
<td>Loop cannot be automatically parallelized. A dependency is carried by variable aliasing or function call.</td>
</tr>
</tbody>
</table>

Listing 4.1: Sample parallelization report from the XL compiler
report is the ID of the file containing the loop, the second column is the line number of the loop, the third column is the per-function ID of the loop, and the fourth column gives the reason why the loop was not automatically parallelized. We argue that this feedback can be improved by (i) including more detail about the reason that parallelization was unsuccessful, and (ii) suggesting to the programmer the guarantee that could be specified to enable parallelization—i.e., it would be the programmer’s job to investigate and confirm the validity of the suggested guarantee, and communicate this to the compiler by including the guarantee in the code.

In the following sections we use the loops from Table 3.3 and Table 3.4 to show how XL C/C++’s programmer guarantees can be used to parallelize the loops and how the support for programmer guarantees can be improved. The next sections are organized according to the reason that the loops were not parallelized: (i) ambiguous pointer/array overlap, (ii) array indirection, and (iii) function call.

4.2 Ambiguous Pointer/Array Overlap

Pointer aliasing and array overlap can cause ambiguous dependences that the compiler cannot resolve, and XL C/C++ outputs the following message in the parallelization report:

Loop cannot be automatically parallelized. A dependency is carried by variable aliasing or function call.

The programmer uses #pragma disjoint to guarantee to the compiler that a set of pointer variables will not alias.

4.2.1 LBM

The LBM benchmark has a run-time parallel loop in lbm.c:186, function LBM_performStreamCollide, shown in Listing 4.2, which copies array elements from
for(i = 0; i < N_CELL_ENTRIES*1300000; i+=N_CELL_ENTRIES) {
    if((srcGrid[FLAGS+i]) & OBSTACLE) {
        dstGrid[C+i] = srcGrid[C+i];
        dstGrid[S+N_CELL_ENTRIES*-100+i] = srcGrid[N+i];
        dstGrid[N+N_CELL_ENTRIES*100+i] = srcGrid[S+i];
        dstGrid[W-N_CELL_ENTRIES+i] = srcGrid[E+i];
        dstGrid[E+N_CELL_ENTRIES+i] = srcGrid[W+i];
    }
    /* ... and more assignments */
}

Listing 4.2: LBM loop (lbm.c:186)

srcGrid to dstGrid. These two arrays are passed into the function as pointers, and so without an expensive inter-procedural analysis (IPA), the compiler would not know whether the arrays overlap or not. XL C/C++ outputs:

Loop cannot be automatically parallelized. A dependency is carried by variable aliasing or function call.

Even though the compiler message does not specify which pointers are causing the dependency, srcGrid and dstGrid are the only two pointers used in the loop, so they must be causing the dependence. The programmer checks the only context where LBM_performStreamCollide is called from to see how srcGrid and dstGrid are defined, and determines that they are two separate, globally-declared arrays. Therefore the programmer can provide the guarantee to the compiler that they do not alias or overlap using:

#pragma disjoint(*srcGrid, *dstGrid)

which leads to successful parallelization by the compiler.

The compiler report message can be made more user-friendly for the programmer to specify guarantees:
Loop cannot be automatically parallelized. An unproven dependency is carried by accesses to *srcGrid and *dstGrid. If these pointers do not alias or overlap, please specify "#pragma disjoint(*srcGrid, *dstGrid)"

The improved message specifies whether the dependence is proven or unproven, includes the pointer variable names involved in the aliasing-induced dependency, and provides a suggestion to the programmer to resolve the aliasing-induced dependency. The improved message should not be difficult to implement: the compiler’s alias analysis pass already has the points-to sets which can be used to specify the pointer variable names in the message. If it is a may-alias relationship, then it is an unproven dependency and not a proven dependency. The compiler also suggests to the programmer that to resolve an aliasing-induced dependency, the #pragma disjoint should be specified if appropriate.

This example is simple enough that the compiler could also generate a guard for the parallelized code, instead of asking the programmer for a guarantee. The guard simply consists of a math calculation to make sure that the minimum and maximum memory addresses accessed with srcGrid and dstGrid do not overlap. This is indeed what the latest version of XL C/C++, V11.1, does to automatically parallelize this loop.

4.2.2 HMMER

The HMMER benchmark has a more interesting and complex pointer example. The loop in fast_algorithms.c:133, function P7Viterbi, shown in Listing 4.3 is mostly parallel, except for lines 141-143 which have loop-carried dependences on dc.

Currently, the XL C/C++ compiler outputs the message:

Loop cannot be automatically parallelized. A dependency is carried by variable aliasing or function call.
for (k = 1; k <= M; k++) {
    mc[k] = mpp[k-1] + tpmm[k-1];
    if ((sc = ip[k-1] + tpim[k-1]) > mc[k]) mc[k] = sc;
    if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k]) mc[k] = sc;
    if ((sc = xmb + bp[k]) > mc[k]) mc[k] = sc;
    mc[k] += ms[k];
    if (mc[k] < -987654321) mc[k] = -987654321;

dc[k] = dc[k-1] + tpdd[k-1];
    if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
    if (dc[k] < -987654321) dc[k] = -987654321;

    if (k < M) {
        ic[k] = mpp[k] + tpmi[k];
        if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
        ic[k] += is[k];
        if (ic[k] < -987654321) ic[k] = -987654321;
    }
} 

Listing 4.3: HMMER loop (fast_algorithms.c:133)

Without the specific details about which pointer aliasing is causing the dependency, it is difficult for the programmer to investigate how to resolve it. In this case, all of the arrays are involved in the aliasing.

The reason that XL C/C++ thinks that there are aliasing-induced dependences is because the arrays mc, mpp, tpmm, ip, tpim, dpp, tpdm, bp, ms, dc, ic, tpmi, tpii, and is point to different regions of big memory buffers that HMMER uses internally.

For example, HMMER allocates a big array of more than L*M integers and assigns it to mx->mmx_mem. Next, the elements of mx->mmx are assigned the addresses of consecutive blocks of M+padding integers of mx->mmx_mem, so that mx->mmx[0]...
Listing 4.4: Parallelizable HMMER loop

= mx->mmx_mem; mx->mmx[1] = mx->mmx_mem+M+padding; // and so on. The arrays mc and mpp are then assigned consecutive elements of mx->mmx, which means that mc and mpp now point to consecutive blocks of M+padding integers. Since the inner P7Viterbi loop does not access more than M elements of mc and mpp, the array accesses will not overlap and do not cause loop-carried dependences.

Therefore, after moving the lines 141-143, which do contain loop-carried dependences, to a different loop, the rest of the original loop can be parallelized by XL C/C++ with the guarantee
that none of the arrays overlap. Listing 4.4 shows the modified loop with the programmer guarantee that XL C/C++ can automatically parallelize.

The same compiler message improvements for LBM can be applied to this loop as well. By specifying which dependences are proven and unproven, the compiler tells the programmer which one to spend more time investigating. Specifying the pointer variable names as well as a suggestion on how to proceed would also save the programmer time:

---
Loop cannot be automatically parallelized. A proven dependence is carried by accesses to *dc.
---

---
---

### 4.3 Array Indirection

When the elements of one array are used to index another array, such as A[B[i]], the enclosing loop can be parallelized if the inner array, B[], has unique values, because no two iterations will use the same element of A[]. XL C/C++ outputs the message for these cases:

---
Loop cannot be automatically parallelized. A dependency is carried by variable "A[]".
---

Since it is difficult for the compiler to know the values of the inner array, these types of dependences are likely unproven and this should be communicated to the programmer so that the programmer knows to double-check this dependency.
Programmers use `#pragma ibm permutation` to guarantee that the inner array only contains unique values.

### 4.3.1 GCC

The GCC benchmark has a run-time parallel loop in `cselib.c:231`, shown in Listing 4.5 that uses array indirection. XL C/C++ outputs the following message:

```
Loop cannot be automatically parallelized. A dependency is carried by variable "(*)varray_head_tag. varray_data_tag.[]0.rns21."
```

The variable `used_regs` is a user-implemented variable-length array that is declared as a static global in the file `cselib.c`, which means it is accessible only in its file. Values are pushed onto `used_regs` in two locations in the file, and in both places there is a check that `REG_VALUES(i)` is not used before pushing `i` onto `used_regs`. Hence, `used_regs` is being used to keep track of the pseudo-registers used and its values are unique. Therefore, the loop can be parallelized.

To parallelize this loop, the programmer guarantees to the compiler that `used_regs->data.u` has unique integers by using the pragma:

```
#pragma ibm permutation(used_regs->data.u)
```

However, the XL C/C++ front-end is not designed to handle such a complex data structure as a union member of a struct member. So the pragma is rejected, and the programmer needs to use

```
#pragma ibm independent_loop
```
to parallelize the loop, which guarantees to the compiler that each loop iteration is independent of the other ones.

The compiler message can be made more user-friendly by specifying that it is an unproven dependency, and suggesting the appropriate programmer guarantee when it sees that the index to the outer array comes from another array look up:

```
Loop cannot be automatically parallelized. An unproven dependency is carried by variable "(*varray_head_tag.
varray_data_tag[]0.rns21.". If the inner array contains unique values, please specify "#pragma ibm permutation(used_regs->data.u)"
```

### 4.4 Function Call

Functions called from within a loop may cause dependences and require expensive IPA to prove otherwise. Even though inlining such functions may help with automatic parallelization, the compiler may deem the function not worth inlining due to its size or other criteria. XL C/C++ outputs the following message when it cannot prove that a function will not cause dependences:

```
Loop cannot be automatically parallelized. Loop contains a call to "function_name" that may have side effects.
```

There are 3 different guarantees that programmers can provide in the presence of function calls, depending on the situation. For the case where the function does not have any side effects, the programmer can use the guarantee `#pragma isolated_call`. For the case where the function does have side effects, but the side effects do not cause loop-carried dependences, the programmer can use the guarantee `#pragma ibm independence_calls`. For the case where the function does have side-effects that cause loop-carried dependences, but the order of the function invocations do not matter, the programmer can use the guarantee `#pragma ibm
critical. The following loops fall under the last 2 cases and will require the respective guarantees.

### 4.4.1 MILC

The benchmark MILC has a run-time parallel loop in `path_product.c:128`, shown in Listing 4.6, which can’t be parallelized due to the function call to `mult_su3_na`. This function performs matrix multiplication of the first two arguments, and stores the resulting matrix in the third argument; hence, it has side effects. XL C/C++ without link-time IPA generates the message:

Loop cannot be automatically parallelized. Loop contains a call to "mult_su3_na" that may have side effects.

and with link-time IPA it will generate the message:

Loop cannot be automatically parallelized. A dependency is carried by variable "(*(aggr#2.rns38.").

which likely refers to the first argument, where the resulting matrices are stored.

This loop can be parallelized if the matrices `gen_pt[0][i]` do not overlap with each other, and if those matrices do not overlap with the `lattice[i].link[7-dir[j]]` and `lattice[i].tempmat1` matrices. The matrix `gen_pt[0][i]` points to the matrix of node i’s neighbour in a certain direction, which in this case points to matrices stored in either
for(i = 0; i < loopend; i++) {
    mult_su3_na(
        (su3_matrix *) (gen_pt[0][i]),
        &(lattice[i].link[7-dir[j]]),
        &(tempmat2t[i])
    );
}

Listing 4.7: MILC loop (path_product.c:85)

tempmat2t[] or tempmat3t[] arrays, and these matrices do not overlap. Moreover, lattice[i].link[7-dir[j]] and lattice[i].tempmat1 are matrix objects, and so they do not overlap with the matrices in tempmat3t or tempmat2t. Hence, the loop is parallel.

To parallelize the loop, the programmer needs to guarantee that the function calls to mult_su3_na, though they may have side effects, the side effects are independent:

#pragma ibm independent calls(mult_su3_na)

After specifying the guarantee and using -qipa=level=2 (link-time IPA), the compiler is able to automatically parallelize the loop.

Since the compiler already determined that the function mult_su3_na has side effects and loop-carried dependences are possible due to the gen_pt[0][i] pointers, the compiler message can be improved by suggesting to the programmer to check if the side effects are independent:

Loop cannot be automatically parallelized. An unproven dependency is carried by variable "(*(aggr#2.rns38).". If the function side effects are independent, please specify "#pragma ibm independent calls(mult_su3_na)".

MILC has another similar run-time parallel loop in path_product:85, shown
in Listing 4.7. The only difference between this loop and the one in Listing 4.6 is that gen pt[0][i] points to matrices in either tempmat3t[] or lattice[neighbor].tempmat1, and the third argument is now a local array of matrices, tempmat2t. These differences do not cause loop-carried dependences; however, specifying the guarantee

```
#pragma ibm independent_calls(mult Su3 na)
```

the compiler still cannot automatically parallelize the loop and instead generates the following report:

```
Loop cannot be automatically parallelized. A dependency is carried by variable aliasing or function call.
```

The message does not mention which pointers are causing aliasing, which makes it difficult for the programmer to investigate what is blocking parallelization. After attempting to use #pragma disjoint and restrict on all the pointers in the function containing the loop and in mult Su3 na, the XL C/C++ compiler still cannot parallelize the loop. Hence, to parallelize the loop the programmer needs to provide the generic guarantee that the loop iterations are independent:

```
#pragma ibm independent_loop
```

To improve the user-friendliness of the compiler message, the compiler should include the names of the pointer variables involved in the aliasing-induced dependence.

### 4.4.2 DEALII

The benchmark DEALII has a quadruple-nested loop at mapping_q1.cc:511, shown in Listing 4.8, which contains a function call to InternalData::derivative. Without link-time IPA, the compiler generates the report message:
for(unsigned int point = 0; point < n_q_points; ++point)
for(unsigned int k = 0; k < data.n_shape_functions; ++k)
for(unsigned int i = 0; i < dim; ++i)
for(unsigned int j = 0; j < dim; ++j)
  data.contravariant[point][i][j] += data.derivative(point+data_set,k)[j] *
  data.mapping_support_points[k][i];

Listing 4.8: DEALII loop (mapping_q1.cc:511)

Loop cannot be automatically parallelized. Loop contains a call to "QProjector<3>::DataSetDescriptor:: operator unsigned int() const" that may have side effects.

With link-time IPA, the compiler generates:

Loop cannot be automatically parallelized. A dependency is carried by variable "(double).rns135."

The function QProjector<3>::DataSetDescriptor:: operator unsigned int() const returns an integer so there are no side effects. The variable (double).rns135 does not show up in the compiler's listing files, and so it is difficult for the programmer to know what is causing a dependency.

In order to determine if the loop is parallel, the programmer finds out that data.contravariant is a vector of disjoint 2-D Tensors, so there is no overlapping of memory accesses; data.derivative() returns an element from the array shape_derivatives, which is a vector of 1-D Tensors, so the function does not have side effects and the memory accesses do not overlap with data.contravariant; and data.mapping_support_points is a vector of 3-D Points which does not overlap with any other vectors. Hence, the loop can be run in parallel.

Since the #pragma ibm guarantees are only supported by the XL C compiler and this is
while (begin!=end)
    contract (* (begin++), *(src++),*(tensor++));

Listing 4.9: DEALII loop (mapping_q.cc:1159)

int len = end - begin;
for (int i = 0; i < len; i++) {
    contract (begin[i], src[i], tensor[i]);
}

Listing 4.10: Normalized DEALII loop (mapping_q.cc:1159)

a C++ benchmark, the programmer has to use OpenMP to parallelize the loop:

#pragma omp parallel for

Another run-time parallel loop in DEALII can be found in mapping_q.cc:1159, shown in Listing 4.9. Without link-time IPA, the compiler generates the report:

Loop cannot be automatically parallelized. Loop contains a call to "void "contract<int(3)>(Tensor<1,3>&,const Tensor<1,3>&,const Tensor<2,3>&)" that may have side effects.

With link-time IPA, the compiler generates the report:

Loop cannot be automatically parallelized. A dependency is carried by variable "(Tensor<1,3>).values[]@0.rns580.6350".

The variable (Tensor<1,3>).values[]@0. rns580.6350 does not appear in the compiler listing file, so it is difficult for the programmer to find out what is blocking parallelization.

The variables begin, src are pointers/arrays of 1-D tensors, end is one element past the end of the begin array, and tensor is a C++ iterator of a vector of 2-D tensors. The contract function essentially multiplies elements of src and tensor and stores the results in begin. For the loop to be parallel, the arrays and vectors cannot overlap, and contract
cannot have side effects that cause loop-carried dependences. Since `begin`, `end` and `src` are parameters of the function, the programmer needs to investigate the call sites to find out where these pointers point to.

There are 22 instances in which the enclosing virtual function could be potentially called, and each time it is called, `begin` points to the beginning of the array used in a C++ vector implementation, `end` points to the end of that array, and `src` points to the first element of a different vector. `tensor` is a local pointer and it points to a different C++ vector. Since vectors cannot overlap, their arrays do not overlap, and the loop is parallel. The loop needs to be normalized first, as shown in Listing 4.10, and, because the more specific programmer guarantees `#pragma ibm`'s are not supported by XL C++, the loop needs to be parallelized using OpenMP:

```
#pragma omp parallel for
```

Moreover, because the more specific programmer guarantees are not supported by the XL C++ compiler, the compiler message cannot be improved to help the programmer by suggesting a guarantee.

### 4.4.3 XALANCBMK

The benchmark XALANCBMK has a run-time parallel loop in `ValueStore.cpp:370`, shown in Listing 4.11, which XL C/C++ cannot parallelize due to the function call to `fScanner->getValidator()->emitError`:

```
Loop cannot be automatically parallelized. Loop contains a call to "xercesc_2_5:XMLValidator:emitterError(xercesc_2_5:XMLValid:Codes, const unsigned short*, const unsigned short*, const unsigned short*, const unsigned short*)" that may have side effects.
```
```c
for (unsigned int i = 0; i < count; i++) {
    FieldValueMap* valueMap = fValueTuples->elementAt(i);
    if (!fKeyValueStore->contains(valueMap) && fDoReportError) {
        fScanner->getValidator()->emitError(
            XMLValid::IC_KeyNotFound,
            fIdentityConstraint->getElementName()
        );
    }
}
```

Listing 4.11: XALANCBMK loop (ValueStore.cpp:370)

Not shown in the compiler report is that the function call to `fKeyValueStore->contains` is also blocking parallelization. The function `fValueTuples->elementAt` does not cause problems because it is inlined.

The function `contains` is where most of the work of the loop is done. It essentially compares every element of one array against all the elements of a second array. The comparison function does not have loop-carried dependences.

The function call to `emitError`, however, does have side effects that cause loop-carried dependences, such as incrementing the error count and generating the error messages. These side effects, though, can happen in any order, as long as only one invocation of `emitError` runs at a time. For example, the final error count will still be the same regardless of the order of the `emitError` calls, and the error messages will still all appear, although they may be in different order, which is acceptable. XL C/C++ currently does not have such a guarantee to specify that a function can be executed in any order as long as only one executes at a time, but this can be emulated with an OpenMP critical region, which ensures that only one thread is executing the function regardless of which thread goes first. Therefore, this loop can be parallelized as shown in Listing 4.12.
Once programmer guarantees are used in a program, they may become incorrect as the program source code is changed. Incorrect guarantees lead to incorrect parallelization and incorrect run-time results. To prevent this, periodic test runs with an instrumented version of the program could be scheduled by the programmer, with instrumentation to verify that the guarantees still hold. For the disjoint pointer guarantee, the instrumentation would consist of pointer comparisons. For the unique array values guarantee, the instrumentation would consist of loops that check the values’ uniqueness. For the function guarantees, instrumented memory accesses would check that no global changes are made, or that if global changes are made, they do not cause loop-carried dependences. Note however that this is only “best-effort” verification, in that unexercised inconsistencies could remain undetected.
4.6 Summary

The compiler can alleviate some of the work required for programmers to use guarantees. The compiler can incorporate loop and dependence profiling into its PDF optimizations, which allows it to filter the program’s loops so that only the loops that are worth parallelizing and that are likely to be parallel are brought to the attention of the programmer. The compiler should also provide as much detail as possible on what is blocking parallelization, so that the programmer can know exactly what ambiguity needs to be verified and resolved through guarantees. Finally, the compiler can suggest the guarantee to the programmer, to further decrease the work of the programmer.
Chapter 5

Performance

For this work we chose to use the SPEC CPU2006 applications because of their abundance of difficult-to-parallelize loops, and so that we may consider a broad range of new programmer guarantees—not because we expect the parallelization of these loops to result in tremendous program speedups. However, for completeness, in this chapter we evaluate the performance impact of the additional loops that were parallelized via programmer guarantees in Chapter 4.

5.1 Methodology

We compare the speedup of the benchmark with the programmer guarantee versus the original benchmark, both compiled with XL C/C++ V10.1 using the most aggressive optimization level (-O5 -qarch=pwr5, which includes loop optimizations, inter-procedural optimizations, whole-program link-time optimizations, and POWER5-specific optimizations), and automatic parallelization (-qsmp). This means that the original benchmark, even though it does not have programmer guarantees, will still have other loops automatically parallelized by XL C/C++ and is not just the sequential version of the benchmark. By doing this, we are evaluating the programmer guarantees against the most aggressively-optimized automatically-parallelized version of the benchmark.

Measurements were made using a quad-core dual-SMT POWER5 1.9GHz processor with
Figure 5.1: Benchmarks that speed up after specifying programmer guarantees compared to the original benchmark, both using the most aggressive optimization level and automatic parallelization.

31GB of RAM running AIX 6.1. Therefore, we ran each benchmark with 1, 2, 4, and 8 threads. For each measurement, the benchmark was run 5 times and the execution time was averaged. The environment variable XLSMPOPTS=yields=0:spins=0 was set so that threads only busy-wait rather than yielding or sleeping when waiting for work, to improve overall performance.

### 5.2 Results

Figure 5.1 shows the performance of the loops that gain a speedup after using programmer guarantees and Figure 5.2 shows the performance of the loops that slow down or that do not speed up. The following sections discuss each benchmark’s performance in more detail.
Figure 5.2: Benchmarks that slow down or don’t speed up after specifying programmer guarantees compared to the original benchmark, both using the most aggressive optimization level and automatic parallelization.

5.2.1 LBM

This is the only benchmark with significant speedup because the key loop takes up 99% of the program run-time, and each loop instance has plenty of work with 1,300,000 iterations and a large loop body. The benchmark achieves 2.5x speedup with 4 threads. Note that the newer version XL C/C++ V11.1 is indeed now able to automatically parallelize this loop with a run-time check, and achieves the same speedup.

5.2.2 MILC

The two loops in MILC, path_product:85 and path_product:128, have a slight speedup of 1.10x and 1.08x respectively with 4 threads because each loop covers 10% of the run-time, and they have plenty of work in more than 1,000 instances and 160,000 iterations each.
5.2.3 XALANCBMK

This benchmark has a speedup of 1.09x because there is enough work in its 11 invocations and 11678 iterations per invocation to offset the thread overhead. Also, there are no errors encountered in the benchmark’s input files, so the OpenMP critical region is never executed and there is no synchronization overhead.

5.2.4 HMMER

This benchmark experiences a 20% slow-down, even though the loop covers 72% of the program run-time, because it has a low iteration-count (100 or 300 depending on the input file used). Furthermore, there is a slow-down with 1 thread because it seems that XL C/C++ is using the guarantee that the pointers do not alias to make bad optimization decisions in other parts of the program. This is demonstrated by parallelizing the same loop with OpenMP rather than the guarantee, which leads to a smaller slow-down of 11%.

5.2.5 GCC

The examined loop in GCC does not experience any speedup because it does not have many iterations (ranges from 0 to 2,767 iterations, but a low average of 5.9), and performs only a small amount of work per iteration (only one assignment). It does not slow down the benchmark because the loop only covers 0.01% of the run-time. Even though this loop is not worth parallelizing due to its low coverage, we include this loop in the dissertation to show how programmer guarantees can be used to parallelize similar loops with ambiguous array indirection in programs where the loop is more important.

5.2.6 DEALII

The loop at mapping_q1.cc:511 does not experience any speedup because it does not have enough work in its 12 iterations, but it does not slow-down either because it has just enough
work in its four levels of loop nestings (12 x 19 x 3 x 3 iterations from outermost to innermost loop) to balance out the thread overhead. The DEALII loop at `mapping_q.cc:1159` experiences a slow-down, to a maximum of 39% with 8 threads, because its 17 iterations have very little work—the `contract` function has a double-nested loop of only 3 x 3 iterations.

### 5.3 Summary

While the XL C/C++ auto-parallelizer is capable of exploiting most of the statically-decidable parallelism in the SPEC CPU2006 benchmark suite, this study was still able to identify a number of interesting loops (namely in LBM, MILC and XALANCBMK), that can be parallelized with programmer guarantees and obtain a program speedup of 2.5x, 1.10x, 1.08x and 1.09x. The other loops in benchmarks HMMER, GCC, and DEALII, though they were not significant enough to result in program speedups, are still interesting examples of loop structures that might appear in other applications and have more impact when parallelized.
Chapter 6

Conclusions

Automatic parallelizing compilers have improved considerably and can parallelize many types of loops. However, there are still many loops that they cannot handle, such as ones with pointers, complex and heap-based data structures, array indirection, and function calls. We use the DProf dependence profiler to analyze the parallelism available in the SPEC CPU2006 benchmark suite that is missed by XL C/C++ V10.1, and found that there are opportunities for using programmer guarantees, guards/run-time checks, TM and TLS to parallelize the benchmarks.

Programmer guarantees have many advantages, but they are difficult to use and are currently limited to use by advanced programmers. We suggest that programmer guarantees be made easier to use by utilizing tools such as loop profilers and dependence profilers to help narrow the selection of loops that the programmer should focus on, and by the compiler providing more specific information about the ambiguity that is blocking parallelization, and even by suggesting a corresponding guarantee that the programmer could make to alleviate the blockage and permit parallelization. We show that we can use programmer guarantees on SPEC CPU2006 loops and obtain a 1.08x, 1.09x, 1.10x and 2.5x speedup in four cases.
6.1 Contributions

In this dissertation, we make the following contributions:

1. we provide the results of an in-depth analysis of the parallelism available in loops in SPEC CPU2006 applications, based on the DProf dependence analysis tool and the IBM XL C/C++ V10.1 parallelizing compiler;

2. through case studies of several loops not successfully parallelized automatically, we suggest improvements to support for programmer guarantees that lead to successful parallelization;

3. we made improvements to IBM’s dependence profiler DProf that enable the profiling of large applications such as the SPEC CPU2006 benchmarks.

6.2 Future Work

In this section we describe potential future avenues of research for improving the support of programmer guarantees.

Firstly, an implementation of the improved support for programmer guarantees is needed. As mentioned in Section 4.1, the XL C/C++ compiler already has features that make it suitable for the improved support of programmer guarantees: profile-directed feedback (PDF), DProf support, and parallelization reports. DProf already has some design features for feeding back its results to XL C/C++, but is only lacking a full implementation. Also, the latest version of the compiler, V11.1, uses XML for its parallelization reports so that it can be easily parsed by third-party tools. This is useful to allow the implementation of programmer guarantee suggestions as a third-party tool instead of part of the compiler.

Secondly, more benchmark suites should be profiled and analyzed to see what other types of programmer guarantees would be required to be able to parallelize even more loop structures. Modern benchmark suites such as PARSEC [4] and Lonestar [20] have many irregular loops
that auto-parallelizers have difficulty parallelizing because they have linked list or graph traversals. However, there is still potential for programmer guarantees. For example, the main work loops in BarnesHut from Lonestar and blackscholes from PARSEC can be parallelized with the function call programmer guarantees described in this dissertation.

Thirdly, it would be interesting to see how programmer guarantees can be applied to Single-Instruction-Multiple-Data (SIMD) vectorization. SIMD vectorization is also blocked by source code ambiguities such as ambiguous pointers, and programmer guarantees can be used to help the compiler auto-vectorize loops.
Bibliography


[24] Mehrara, M., Hao, J., Hsu, P.-C., and Mahlke, S. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory.


Appendix A

Loops of Interest in SPEC CPU2006

A.1 Loops of Interest for Manual Parallelization

Loops that are of interest for manual parallelization have a high percentage of parallel instances to maximize speedup ($\geq 50\%$), a low number of dependences to minimize the parallelization effort ($\leq 5$), and a low frequency of dependences to maximize speedup ($\leq 10\%$). The SPEC CPU2006 loops that meet this criteria and have at least 2 iterations are listed in Table A.1.

A.2 Loops of Interest for Run-time Checks

Loops of interest for compiler developer looking into improving support for run-time checks have a high percentage of parallel instances to maximize speedup ($\geq 50\%$), and a low number of dependences to minimize the overhead of the check ($\leq 5$). The SPEC CPU2006 loops that meet this criteria and have at least 2 iterations are listed in Table A.2.

A.3 Loops of Interest for Speculative Parallelization

Loops of interest for compiler developers working on speculative parallelization have a high percentage of parallel instances to maximize speedup($\geq 50\%$), a low number of dependences
(≤ 5) and a low frequency of occurring dependences (≤ 10%) to minimize conflicts and rollbacks, and a high average independence window size to maximize the number of threads (≥ 5). The SPEC CPU2006 loops that meet this criteria and have at least 2 iterations are listed in Table A.3.
<table>
<thead>
<tr>
<th>Loop</th>
<th>Benchmark</th>
<th>Run-time</th>
<th>Invocations</th>
<th>Iterations</th>
<th>Parallel Instances</th>
<th>Number of Dependences</th>
<th>Dependence Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>fe.cc:489:10</td>
<td>dealII</td>
<td>30.16%</td>
<td>175728</td>
<td>3</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>include/base/tensor.h:502:1</td>
<td>dealII</td>
<td>31.41%</td>
<td>1687241000</td>
<td>3</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>mapping_q.cc:1159:1</td>
<td>dealII</td>
<td>36.36%</td>
<td>99091940</td>
<td>17</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>mapping_q1.cc:511:2</td>
<td>dealII</td>
<td>30.50%</td>
<td>5340012</td>
<td>12</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>mapping_q1.cc:513:4</td>
<td>dealII</td>
<td>26.41%</td>
<td>1342991000</td>
<td>3</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>mapping_q1.cc:514:5</td>
<td>dealII</td>
<td>11.46%</td>
<td>4028972000</td>
<td>3</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>step-14.cc:3348:3</td>
<td>dealII</td>
<td>11.79%</td>
<td>175728</td>
<td>6</td>
<td>89.94%</td>
<td>1</td>
<td>2.19%</td>
</tr>
<tr>
<td>lbm.c:186:1</td>
<td>lbm</td>
<td>98.96%</td>
<td>3000</td>
<td>1300000</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>m_matna.c:31:1</td>
<td>milc</td>
<td>17.41%</td>
<td>4608000000</td>
<td>3</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>m_matnn.c:73:1</td>
<td>milc</td>
<td>15.06%</td>
<td>461744000</td>
<td>3</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>path_product.c:145:6</td>
<td>milc</td>
<td>10.50%</td>
<td>1116</td>
<td>160000</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>path_product.c:91:10</td>
<td>milc</td>
<td>10.75%</td>
<td>1028</td>
<td>160000</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>ValueStore.cpp:370:1</td>
<td>xalanbmk</td>
<td>25.09%</td>
<td>11</td>
<td>11678</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table A.1: Loops of interest for manual parallelization
<table>
<thead>
<tr>
<th>Loop</th>
<th>Benchmark</th>
<th>Run-time</th>
<th>Invocations</th>
<th>Iterations</th>
<th>Parallel Instances</th>
<th>Number of Dependences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Way2_.cpp:65:4</td>
<td>astar</td>
<td>23.72%</td>
<td>1965981000</td>
<td>3</td>
<td>76.17%</td>
<td>1</td>
</tr>
<tr>
<td>fe.cc:489:10</td>
<td>dealII</td>
<td>30.16%</td>
<td>175728</td>
<td>3</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>include/base/tensor.h:502:1</td>
<td>dealII</td>
<td>31.41%</td>
<td>1687241000</td>
<td>3</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>mapping_q.cc:1159:1</td>
<td>dealII</td>
<td>36.36%</td>
<td>99091940</td>
<td>17</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>mapping_q1.cc:511:2</td>
<td>dealII</td>
<td>30.50%</td>
<td>5340012</td>
<td>12</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>mapping_q1.cc:513:4</td>
<td>dealII</td>
<td>26.41%</td>
<td>1342991000</td>
<td>3</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>mapping_q1.cc:514:5</td>
<td>dealII</td>
<td>11.46%</td>
<td>4028972000</td>
<td>3</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>step-14.cc:3348:3</td>
<td>dealII</td>
<td>11.79%</td>
<td>175728</td>
<td>6</td>
<td>89.94%</td>
<td>1</td>
</tr>
<tr>
<td>lbm.c:186:1</td>
<td>lbm</td>
<td>98.96%</td>
<td>3000</td>
<td>1300000</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>m_mat_na.c:31:1</td>
<td>milc</td>
<td>17.41%</td>
<td>460800000</td>
<td>3</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>m_mat.nn.c:73:1</td>
<td>milc</td>
<td>15.06%</td>
<td>461744000</td>
<td>3</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>path_product.c:145:6</td>
<td>milc</td>
<td>10.50%</td>
<td>1116</td>
<td>1600000</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>path_product.c:91:10</td>
<td>milc</td>
<td>10.75%</td>
<td>1028</td>
<td>1600000</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>ValueStore.cpp:370:1</td>
<td>xalancbmk</td>
<td>25.09%</td>
<td>11</td>
<td>11678</td>
<td>100%</td>
<td>0</td>
</tr>
</tbody>
</table>

Table A.2: Loops of interest for run-time checks
## Table A.3: Loops of interest for speculative parallelization

<table>
<thead>
<tr>
<th>Loop</th>
<th>Benchmark</th>
<th>Run-time</th>
<th>Invocations</th>
<th>Iterations</th>
<th>Parallel Instances</th>
<th>Number of Dependencies</th>
<th>Dependence Frequency</th>
<th>Average Independence Window Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>fe.cc:489:10</td>
<td>dealII</td>
<td>30.16%</td>
<td>175728</td>
<td>3</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
<td>∞</td>
</tr>
<tr>
<td>include/base/tensor.h:502:1</td>
<td>dealII</td>
<td>31.41%</td>
<td>1687241000</td>
<td>3</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
<td>∞</td>
</tr>
<tr>
<td>mapping_q.cc:1159:1</td>
<td>dealII</td>
<td>36.36%</td>
<td>99091940</td>
<td>17</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
<td>∞</td>
</tr>
<tr>
<td>mapping_q1.cc:511:2</td>
<td>dealII</td>
<td>30.50%</td>
<td>5340012</td>
<td>12</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
<td>∞</td>
</tr>
<tr>
<td>mapping_q1.cc:513:4</td>
<td>dealII</td>
<td>26.41%</td>
<td>1342991000</td>
<td>3</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
<td>∞</td>
</tr>
<tr>
<td>mapping_q1.cc:514:5</td>
<td>dealII</td>
<td>11.46%</td>
<td>4028972000</td>
<td>3</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
<td>∞</td>
</tr>
<tr>
<td>step-14.cc:3348:3</td>
<td>dealII</td>
<td>11.79%</td>
<td>175728</td>
<td>6</td>
<td>89.94%</td>
<td>1</td>
<td>2.19%</td>
<td>5.69</td>
</tr>
<tr>
<td>lbm.c:186:1</td>
<td>lbm</td>
<td>98.96%</td>
<td>3000</td>
<td>1300000</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
<td>∞</td>
</tr>
<tr>
<td>m_mat na.c:31:1</td>
<td>milc</td>
<td>17.41%</td>
<td>460800000</td>
<td>3</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
<td>∞</td>
</tr>
<tr>
<td>m_mat nn.c:73:1</td>
<td>milc</td>
<td>15.06%</td>
<td>461744000</td>
<td>3</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
<td>∞</td>
</tr>
<tr>
<td>path_product.c:145:6</td>
<td>milc</td>
<td>10.50%</td>
<td>1116</td>
<td>1600000</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
<td>∞</td>
</tr>
<tr>
<td>path_product.c:91:10</td>
<td>milc</td>
<td>10.75%</td>
<td>1028</td>
<td>1600000</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
<td>∞</td>
</tr>
<tr>
<td>ValueStore.cpp:370:1</td>
<td>xalancbmk</td>
<td>25.09%</td>
<td>11</td>
<td>11678</td>
<td>100%</td>
<td>0</td>
<td>0%</td>
<td>∞</td>
</tr>
</tbody>
</table>