If you can't explain it simply, you don't understand it well enough.

– Albert Einstein.

University of Alberta

DO INPUTS MATTER? USING DATA-DEPENDENCE PROFILING TO EVALUATE THREAD LEVEL SPECULATION IN THE BLUEGENE/Q

by

Arnamoy Bhattacharyya

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Fall 2013
Edmonton, Alberta

To Late Mohan Lal Bandyopadhyay
Dadu, wish you were here.

Abstract

In the era of many-core architectures, it is necessary to fully exploit the maximum available parallelism in computer programs. Thread Level Speculation (TLS) is a hardware/software technique that guarantees correct speculative parallel execution of the program even in the presence of *may* dependences. This thesis investigates the variability of dependence behaviour of loops across program inputs with the help of data-dependence profiling. This thesis also presents *SpecEval*, a new automatic speculative parallelization framework that uses single-input data-dependence profiles to evaluate the TLS hardware support in the IBM's BlueGene/Q (BG/Q) supercomputer. A performance evaluation of TLS applied along with the traditional automatic parallelization techniques indicates that various factors such as: the number of loops speculatively parallelized and their coverage, mispeculation overhead due to dependences introduced from function calls inside loop body, increase in L1 cache misses due to *long running* (LR) mode in BG/Q and dynamic instruction path length increase impact the performance of TLS.

Acknowledgements

First and foremost, I would like to thank my supervisor Dr. José Nelson Amaral for his constant motivation and help during the course of my research. I would also like to thank Jeff Hammond and Hal Finkel from the Argonne National Laboratory for giving access to the IBM's BlueGene/Q (BG/Q) machine and helping out whenever I had questions. I would like to thank Paul Berube for helping out with the understanding of LLVM and Combined-Profiling, Matthew Gaudet for helping with information about BG/Q.

On a personal note, I would like to thank all the lovely persons who helped me settle in a foreign country and made me feel at home — Anupam *da*, Satarupa *di*, Sanjay *da*, Chandan, Dibyo, Sudarshan, Lokesh, David, Zohaib, Jeeva, the CS graduate group — Amrit Pal, Kyrylo Shegada, Alejandro Ramirez, Eric Tsala, Pierre Rosado and Ryan Kiros.

I would specially thank my uncle, Sumit *kaku* for encouraging me to pursue higher studies.

Lastly, I would like to thank my mom (*maam*), Mandira Bhattacharyya and dad (*bai*), Arun Kumar Bhattacharyya for opening up the bigger world to me.

Table of Contents

List of Tables

List of Figures

Acronyms

TLS	Thread Level Speculation
LLVM	Low Level Virtual Machine
CFG	Control Flow Graph
SPEC	Standard Performance Evaluation Corporation
TM	Transactional Memory
BG/Q	BlueGene/Q
ARB	Address Resolution Buffer
TLB	Translation Lookaside Buffer
MMU	Memory Management Unit
CMP	Chip Multiprocessor

Chapter 1

Introduction

With the advent of multi-threaded and multi-core architectures, it is a challenge to effectively utilize them to improve the performance of general-purpose or scientific applications. Automatic compiler parallelization techniques are being developed and are found to be useful for many of these architectures. However, traditional auto-parallelization techniques can not be effectively applied to general-purpose integer-intensive applications that have complex control flow and excessive pointer accesses because there is no guarantee of the correct execution of the program. Therefore the existing auto-parallelizers follow a conservative approach and synchronize all the program parts (mainly loops) with potential dependences.

These auto-parallelization frameworks can only parallelize a loop when the compiler can prove, using compile-time and/or run-time techniques, that the parallel execution of the loop will not affect the correctness of the program. This constraint often restricts the maximum parallelism that can be extracted from the loops in a program. An auto-parallelization framework uses the result from a compile-time/run-time *dependence analysis* to make a decision about parallelizing a loop so that all executions of the program are correct. *Dependence-analysis* techniques work by checking whether the same address may be accessed (loaded from or stored into) in different iterations of a loop. If two different iterations of the loop access the same memory address, the loop is said to be dependent and it is not parallelized.

Many static-dependence analysis techniques exist that can be used by a parallelizing framework to determine which loops are parallelizable. If the compiler can not determine, at compile time, whether there will be a dependence at run-time, *dependence profiling* is used to determine if any dependence is occurring at run-time [24, 14, 25, 8, 52, 46]. In dependence profiling, the memory accessed by the possibly dependent load/store instructions are observed for any dependences. The result of this *training* run of the program is kept in a profile file. To make parallelizing decisions, the profile file is consulted.

Based on a previous claim that a loop's dependence behaviour does not change with respect to program inputs [14, 25, 45], there is prior work on the collection of data-dependence profile statistics for a single profiling run of a program [46]. But there has not been an extensive study to find

if the claim is really true. Berube *et al.* showed that a program's behaviours change for different inputs [4]. Therefore, if there is an application that has a loop with varied dependence behaviour across inputs, profile collected from a single training run is often not sufficient to perform a cost analysis and find beneficial loops for speculative execution. Berube *et al.* presented a method called Combined-Profiling (CP) to statistically combine the profiles obtained from training runs with different inputs. CP uses histograms to store combined-profiles. By consulting these histograms, the compiler can determine the probability of the desired behaviour (probability of a taken edge in the Control Flow Graph (CFG), and the probability of a loop being dependent/ independent) of the program or program part (loop or function). But to ensure correct execution, there should be a guarantee that the parallel execution of the loop will not affect the correctness of the program even if the dependence materializes during run-time.

Thread-level speculation (TLS) [43] is a technique that guarantees correctness even if there is a dependence inside the loop. The hardware support for speculative threads eases the burden of creating parallel threads by the programmers and compilers. Thread-Level Speculation (TLS) has been used to exploit parallelism in sequential applications that are difficult to parallelize using traditional parallelization techniques. For example, a loop that contains an inter-thread data-dependence due to loads and stores through pointers, cannot be parallelized using traditional compilers. But with the help of TLS, the compiler can parallelize this loop speculatively with the support of the underlying hardware to detect and enforce inter-thread data-dependences at run-time.

There has been a significant amount of research on how to automatically extract speculative parallelism from programs and on how to support TLS in hardware [9, 43, 47, 20, 36, 32, 13]. One way to find profitable loops from the program to be executed in parallel is to look at the probability of the loop being dependent and perform a cost analysis that takes into consideration the dependence probability along with other factors (*e.g.* the execution time of the loop as compared to the whole execution time of the program etc.) that effect the speculative execution of the loop.

IBM's BlueGene/Q (BG/Q) is a supercomputer that has TM and TLS support [21]. BG/Q supports these two hardware features with the help of versioning cache. The hardware stores a consistent program state in the L2 cache and, in case of mispeculation, the program is rolled back to a previous consistent state.

With TLS hardware in hand, this research makes the following contribution:

- A detailed experimentation with 57 different benchmarks, used in previous TLS research, to study the previous claim that for most applications, a loop's dependence behaviour does not change based on inputs.

- An implementation of CP is done to combine the data-dependence profiles obtained from different training runs of applications that have loops with varied dependence behaviour across inputs. The combined-profile stores as much dependence information as is necessary to perform a profitability analysis to find loops that are speculation candidates. TLS performance of

an application (Convex 2D Hull), that has a loop with varied dependence behaviour, is studied.

- A study on the performance impact of applying TLS along with the existing auto-parallelization techniques — automatic SIMD and OpenMP parallelizer for the SPEC2006 and PolyBench/C benchmarks.

- A detailed study of different factors that impact the speculative execution of the loops of the aforesaid benchmarks in BG/Q is performed. This work is the first evaluation of the TLS implementation in BG/Q.

The rest of the thesis is organized as follows - Chapter 2 gives background information on TLS and data-dependence profiling. Chapter 3 demonstrates the proposed CP methodology for data-dependence profiling as implemented in the LLVM compiler. This chapter also includes a discussion on the observed non-variability of the dependence behaviour of loops in 57 different benchmarks from the SPEC2006, BioBenchmark, NAS and PolyBench/C benchmark suites. A study is also included that demonstrates the impact of varied percentages of dependences in a loop of an application on the TLS performance of that application (Convex 2D Hull). Chapter 4 presents SpecEval that uses some simple heuristics to find profitable loops for speculative execution. This chapter also gives a performance comparison of the speculative version of the benchmarks with the parallel version generated by the automatic OpenMP parallelizer of Polly. The effect of data-dependence profiling to reduce the mispeculation overhead and filtering cold loops to reduce the thread-creation overhead on programs' performance is also studied. In Chapter 5, SpecEval uses the dependence analyzer of LLVM and SpecEval is used to evaluate the effect of applying TLS along with the existing parallelization techniques (SIMDized and OpenMP parallelization) of the *bgxlc_r* compiler to the SPEC2006 and PolyBench/C benchmarks. Chapters 7 and 6 present the conclusion and related work of this research.

Chapter 2

Background

This chapter presents the necessary background information on data-dependence, feedback-directed optimization (FDO), combined-profiling and TLS. In Section 2.1, various kinds of data-dependences that may be present in the program, are explained. The section also includes motivating examples where *may dependences* inside loops materialize or do not materialize based on program inputs. Section 2.2 gives the background information on Feedback Directed Optimization (FDO), describing different steps of FDO and *dependence profiling*. Section 2.3 gives background information on Combined Profiling and Section 2.4 describes different concepts related to TLS, with a discussion on special hardware features that are necessary to support TLS. The section also describes the features of BG/Q that enables TLS and the TLS-specific compiler information in the BG/Q. Lastly, Section 2.5 gives information about the different useful passes and tools in LLVM that are used in the research.

2.1 Data-Dependence

A data-dependence occurs in a computer program when two program statements access the same memory location and at least one of the accesses is a write. If there are two statements S1 and S2, S2 depends on S1 when

-

$$[R(S1) \bigcap W(S2)] \bigcup [W(S1) \bigcap R(S2)] \bigcup [W(S1) \bigcap W(S2)] \neq \phi$$

Where $R(S)$ is the set of memory locations read by a statement S, and $W(S)$ is the set of memory locations written by statement S. Dependences exist when there is a feasible run-time execution path from $S1$ to $S2$. This is called the *Bernstein Condition*, named after A. J. Bernstein [2].

Dependences are categorized in the following manner -

- **Flow (data) dependence** $W(S1) \bigcap R(S2) \neq \phi$: $S1$ writes something read by $S2$
- **Anti-dependence** $R(S1) \bigcap W(S2) \neq \phi$: $S1$ reads something before $S2$ overwrites it.

- **Output dependence** $W(S1) \bigcap W(S2) \neq \phi$: Both $S1$ and $S2$ write to the same memory location.

Dependence in Loops

Data-dependence in loops occur when the same memory location is accessed (write/read) by different statements within the same iteration or by different instances of the same statement or different statements between different iterations of the loop. Based on the different kinds of dependences that might occur inside loops, the dependences in loops can be classified as the following two types:

- **Loop Independent**: Dependence between statements executed within the same loop iteration.
- **Loop Carried**: When the same address is accessed by statements executed in different iterations of the loop.

Using static dependence-analysis techniques, the compiler determines, at compile-time, whether there is any dependence among the statements of a loop. Generally the determination of dependences in a loop is based on alias analysis. Two pointers in the program are called *aliases* when they refer to the same variable. The alias analysis performed by the compiler can return three types of aliasing relationships: *must*, *may* and *no* alias. Using the *must* and *no* aliasing information, the compiler can determine statically whether a loop can be parallelized or not. But for *may* aliases, the parallelizability of the loop is not statically provable. Figure 2.1 gives an example of a program where the alias relationship changes based on the input to the program. The dependence relationship of the second *for* loop inside *main* cannot be determined by the compiler and is reported as *may* dependence.

Some dependence analysis techniques use algebraic representations to find out the dependence behaviour of a loop [26, 41, 39]. These dependence analysis techniques operate on array subscripts. But they require the array subscripts to be an *affine* function of the loop induction variables. If the subscript is a nested array access (i.e., the subscript is an element of another array) or is determined by a function call, compilers can not prove the dependence at compile time and they report *may* dependence. Figure 2.2 demonstrates a program where the dependence relation of the statement inside the second *for* loop of the *main* function is not provable at compile time because the subscript of the array is a function call makes the loop either dependent or parallel based on user input.

Due to compile-time constraints and the lack of proper algebraic representation of loops, traditional program transformations can not adapt the *schedule* of statement instances of a program. If the data-dependences in the loops are non-uniform [33] or if the profitability of a transformation is unpredictable, compilers typically cannot apply loop transformations.

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Please enter a choice: 0/1 \n");
    int choice;
    scanf("%d", &choice);
    int x[20], y[20];
    int *a, *b;
    int m;
    /*initialization array*/
    for (m = 0; m < 20; m++)
    {
        x[m] = 5;
        y[m] = 20;
    }
    a = x;
    b = y;
    /*the references 'a' and 'b' changes based on input*/
    if (choice == 0)
        a = b;
    for (m = 4; m < 20; m++)
        /*this statement either depends on itself or not
        based on input */
        a[m] = b[m-2] + 4;

    printf ("Tenth element of array 'x' is %d\n", x[9]);
    return 1;
}
```

Figure 2.1: Sample code that shows a loop where the alias relationship cannot be determined at compile time. The second *for* loop in *main* can be either parallel or not based on the user input *choice*.

2.1.1 Polyhedral Compilation

The Polyhedral model offers a flexible and expressive representation for loop nests with statically predictable control flow. Such loop nests in the program are called *static control parts* (SCoP) [15, 18]. Within a function body, a static control part (SCoP) is a maximal set of consecutive statements without *while* loops, where loop bounds and conditionals may only depend on invariants within this set of statements. These invariants include symbolic constants, formal function parameters and surrounding loop counters: they are called the global parameters of the SCoP, as well as any invariant appearing in some array subscript within the SCoP. The control and data flow information of SCoPs are represented as the following three components:

- **Iteration domains** - Iteration domains are used to consider each dynamic instance of a statement through a set of affine inequalities. Each dynamic instance of a statement S is denoted by a pair (S, i) where i is the iteration vector and it contains values of the surrounding loop indices of the statement. If the loop bounds are affine expressions of outer-loop indices and global parameters, then the set of all iteration vectors i for a given statement S can be represented by a polytope $D_s = \{i | D_s \times (i, g, 1)^T \geq 0\}$. It is called the iteration domain of the statement S, where g is the vector containing all the global parameters of the loop whose

```c
#include <stdio.h>
int change (int, int);
int main (int argc, char * argv[])
{
    printf("Please enter your choice: 0/1 \n");
    int choice;
    scanf("%d", &choice);
    int x[20];
    int m;
    /*initialization array*/
    for(m = 0; m < 20; m++)
        x[m] = 5;
    for(m = 4; m < 18; m++)
        /*this statement either depends on itself
         * or not based on user input*/
        x[m] = x[change(m,choice)] + 4;

    printf ("Tenth element of array 'x' is %d\n", x[9]);
    return 1;
}
int change ( int subscript, int choice)
{
    if (choice == 0)
        return (subscript - 2);
    else
        return 19;
}
```

Figure 2.2: Sample code that shows a loop where the alias relationship can not be determined at compile time. The second *for* loop in *main* can be either parallel or not based on the user input *choice*. Here the array subscript is a function call and dependence analysis techniques report the dependence as a *may* dependence.

dimensionality is d_g.

- **Memory access functions** - These functions are used to represent the locations of data the statements are accessing. In SCoPs, memory accesses are normally performed on array references. For each statement S two different sets are defined - R_s and W_s of (M, f) pairs. Each pair represents a reference to a variable M being written or read by a statement S. f is the access function that maps iteration vectors in D_s to the memory locations in M.

- **Scheduling function** - The set of statement instances that are to be executed dynamically are defined by their iteration domains. But the execution order of each statement instance with respect to other statement instances are not described by this algebraic structure [16]. A scheduling function $_S$ is defined for each statement S that maps instances of S to totally ordered multidimensional timestamps (vectors).

Polyhedral dependence analysis uses the above three components to find dependences and apply correct transformations to the SCoPs. Polyhedral Dependence analysis is used to find may dependences in Chapter 4.

If there are *may* dependences inside a loop as reported by one of these dependence analysis techniques, the loop can not be parallelized. Data-dependence profiling, one example of Feedback-

Directed Optimization (FDO), is necessary to find out if the *may* dependences materialize during runtime. The following section describes FDO and data-dependence profiling.

2.2 Feedback-Directed Optimization

Feedback-directed optimization (FDO) is a compiler optimization technique where the behaviour of the program is observed for some *training* run. The information gathered in the *profiling* (training) run is kept in a *profile* file. The profile file is read by the compiler in a subsequent analysis pass where different optimizations can be performed based on the profile information. A typical feedback-directed optimization is done in the following steps:

- **Instrumentation**

  Compiler optimizations work on some Internal Representation (IR) of the source code. The compiler front end converts the high-level constructs of the source code to a language independent IR. An instrumentation pass in FDO instruments the IR by adding calls to some profiling library functions so that the behaviour of interest can be captured.

- **Profiling**

  The instrumented bitcode is run with some training input and the output is stored in a profile file.

- **Optimization**

  After the profiling run, an optimization pass is executed that takes both the original intermediate representation (called 'bitcode' in terms of LLVM) and the profile file as inputs and applies some code transformation based on the profiling data. As a result of this pass, an optimized (may not be optimal) version of the bitcode is produced that in turn is translated to an executable by the back end.

2.2.1 Dependence Profiling

Different behaviours of the program can be captured with the help of profiling. Some examples are *edge profiling* that records the execution frequency of the edges in the Control Flow Graph (CFG) of the program, *path profiling* that records the execution frequency of the different execution paths in the Call graph of the program, *execution profile* that records the execution time, or other resource consumption, of the program that contains the execution time, resource consumption etc. by various routines in the program, *value profile* that records the frequency of occurrence of values during the execution of the program, *power profiling* that records the power consumption by various components of the program.

Dependence-profiling stores information about the different memory locations referenced by program regions. A program region may be any single-entry/single-exit region, including a single

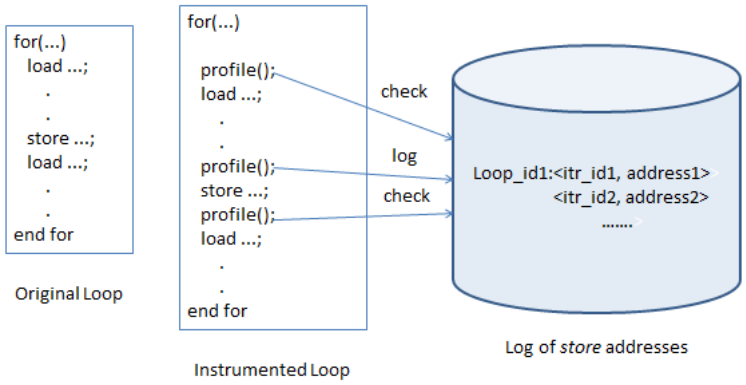

Figure 2.3: An example of a dependence profiler.

statement, an entire loop, or a whole procedure. For loops, the memory address accessed, the type of access (read/write) and the iteration ID for a particular loop are recorded to find dependences. The technique works by putting memory addresses accessed by *loads* and *stores* in loop iterations in a table (typically a hash table for fast lookup). Whenever a new *load* or *store* is executed, the table is searched to see whether the same address was accessed in a previous iteration. If the same memory was accessed before, a new dependence is found. Figure 2.3 demonstrates dependence profiling to find the RAW dependences in a loop. Calls to the profiling library functions are inserted before every load and store instructions in the bitcode of the loop. For stores, the function logs a tuple $< iteration\_id, memory\_address >$ in memory. Whenever a load is executed, the function checks in the log for a matching memory address to find a dependence.

2.2.2 Cost of profiling

Profiling comes with a cost. There are two types of overheads that may arise due to profiling:

1. Space Overhead
2. Time Overhead

For dependence profiling, the memory overhead is huge if the memory accesses for the *may dependent* statements across the *whole* iteration space of the loop is stored. Therefore, to reduce this overhead, the dependence analysis techniques normally work on *loop samples* (some considerable portion of the iteration space) to get an approximation of the dependence behaviour. Moreover, some compression techniques can also be applied to reduce the storage required for profiling.

The timing overhead is optimized by either a fast lookup algorithm (*e.g.* hash table) and/or by creating smaller search space (with the use of access sets [25]). For Just-in-time (JIT) compilers that use run-time profiling, time overhead is a major issue. While in the case of *off-line* profiling (profiling done in a separate profiling run), the time is not a major issue. But still the profiling time should be reasonable.

2.3 Combined-Profiling

A program's behaviour may change based on the inputs to the program. For example, in the programs of Figures 2.1 and 2.2, for an input of '0', the second *for* loop inside *main cannot* be executed in parallel while for an input of '1', the loop can be parallelized.

FDO has not achieved widespread use by compiler users because the selection of a data input to use for profiling that is representative of the execution of the program throughout its lifetime is difficult. For large and complex programs dealing with many use cases and used by a multitude of users, assembling an appropriately representative workload may be a difficult task. Picking one training run to represent such a space is far more challenging, or potentially impossible, in the presence of mutually-exclusive use cases. Moreover, user workloads are prone to change over time. Performance gains today may not be worth the risk of potentially significant performance degradation in the future.

Berube *et al.* [4] proposed a method called CP that eases the burden of training-workload selection while also mitigating the potential for performance degradation. Using their methodology, there is no need to select a single input for training because data from any number of training runs can be merged into a combined-profile. More importantly, CP captures variations in execution behaviour between inputs. The distribution of behaviours can be queried and analyzed by the compiler when making code transformation decisions.

The profile of a program records information about a set of program behaviours. A program behaviour B is a (potentially) dynamic feature of the execution of a program. The observation of a behaviour B at a location l of a representation of the program is denoted B_l A behaviour B is quantified by some metric $M(B)$ as a tuple of numeric values. A monitor $R(B, l, M)$ is injected into a program at every location l where the behaviour B is to be measured using metric M . At the completion of a training run, each monitor records the tuple $< l, M(B_l) >$ in a raw profile. In case of dependence profiling, the *behaviour* to be observed is whether the loop is independent or not. A *monitor* is inserted inside every loop of the program and the *metric* is the number of independent/ dependent executions.

CP stores the profile information with the help of histograms. Histograms are built in an incremental fashion in CP, thus removing the cost of storing multiple profile files that might increase the storage cost. In general, updating produces a new histogram in 2 steps:

1. Determine the range of the combined data. Create a new histogram with this range.

2. Proportionally weight the bins of the new histogram.

The combination of two histograms H_1 and H_2 into a new histogram H_3 is illustrated in Figure 2.4. The range of H_3 is simply the minimum encompassing range of the ranges of H_1 and H_2 : $[min(100, 85), max(150, 125)] = [85, 150]$.

This range is divided into the same number of bins as were present in the original histograms.

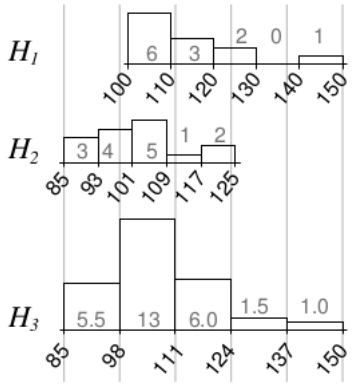

Figure 2.4: Combining histograms. H_1 has a bin width of 10 and a total weight of 12; H_2 has a bin width of 8 and a total weight of 15. The combined histogram H_3 has a bin width of 13 and a total weight of 27 (figure taken from Paul Berube's PhD thesis [3]).

The weight of a bin b_i of H_3 is given by the weights of the bins of H_1 and H_2 that overlap the range of bi multiplied by the overlapping proportion. For example, let b_3 be the third bin of H_3 in Figure 2.4. In H_1 the bin width is 10, and in H_2 the bin width is 8. The weight of b_3 in H_3 is calculated as follows-

$$W_{b3}(H1) = (((120 - 111) \div 10) \times 3) + (((124 - 120) \div 10) \times 2) = 3.5$$

$$W_{b3}(H2) = (((117 - 111) \div 8) \times 1) + (((124 - 17) \div 8) \times 2) = 2.5$$

$$W_{b3}(H3) = (3.5 + 2.5) = 6.0$$

The next section describes the different concepts related to TLS.

2.4 Thread Level Speculation

2.4.1 Overview

1. **Predecessor and Successor Threads**

   Under the thread-level speculation (also called speculative parallelization) approach, sequential sections of code are speculatively executed in parallel hoping not to violate any sequential semantics. The control flow of the sequential code imposes a total order on the threads. At any time during execution, the earliest thread in the program order is non-speculative while the later ones can be speculative. The terms *predecessor* and *successor* are used to relate threads in this total order. In most schemes a violation rolls the execution back to the start of the thread, but some proposals in the literature use periodic checkpointing of threads such that upon a squash it is only necessary to roll the execution back to the closest safe checkpointed state. When the execution of a nonspeculative thread completes it commits and the values it

generated can be moved to safe storage (usually main memory or some shared higher-level cache). At this point its immediate successor acquires non-speculative status and is allowed to commit. When a speculative thread completes it must wait for all predecessors to commit before it can commit.

2. **Inter Thread Data-Dependence**

   Data-dependences are typically captured by monitoring the data written and the data read by individual threads. A data-dependence violation occurs when a thread writes to a location that has already been read by a successor thread. Dependence violations lead to the abortion of the thread that involves discarding the side effects produced by the thread being squashed.

3. **Buffering of States**

   Stores performed by a speculative thread generate a speculative state that cannot be merged with the safe state of the program because this may lead to incorrect results. Such a state is stored separately, typically in the cache of the processor. They are not written back to memory. In case of a violation is detected, the state is discarded from the cache. Also if a speculative thread overflows its speculative buffer the thread must stall and wait to become non-speculative. When the thread becomes non-speculative, the state is allowed to propagate (commit) to memory. When a non-speculative thread finishes execution, it commits. Committing informs the rest of the system that the state generated by the task is now part of the safe program state.

4. **Data Versioning**

   A thread has at most a single version of any given variable. However, different speculative threads running concurrently in the machine may write to the same variable and, as a result, produce different versions of the variable. Such versions must be buffered separately. Moreover, readers must be provided the correct versions. Finally, as threads commit in order, data versions need to be merged with the safe memory state also to ensure correctness.

5. **Multi-Versioned Caches**

   A cache that can hold state from multiple tasks is called multi-versioned [9, 47, 43].

   There are two performance reasons why multi-versioned caches are desirable: they avoid processor stalls when there is imbalance between tasks.

Speculative threads are usually extracted from either loop iterations or function continuations, without taking into consideration possible data-dependence violations. The compiler marks these structures with a fork-like spawn instruction, so that the execution of such an instruction leads to a new speculative thread. The parent thread continues execution as normal, while the child thread is mapped to an available core. For loops, spawn points are placed at the beginning of the loop body, so that each iteration of the loop spawns the next iteration as a speculative thread. Threads formed from

iterations of the same loop (and that, thus, have the same spawn point) are called *sibling threads*. For function calls, spawn points are placed just before the function call such that the non-speculative thread proceeds to the body of the function, and a speculative thread is created from the functions continuation [53].

Two different overheads should be considered when running loops speculatively in parallel. The first overhead comes from buffering the state of the program before the start of speculative execution so that the program can be rolled back to a previous consistent thread. If the computation done inside a thread is not large enough to overcome this cost, there is no benefit from speculation.

The overhead comes from the cost of mispeculation. If there is actual dependence occurring between threads, the younger thread is squashed. After a number of retries, the loop becomes sequential. Thus if the probability is high that the loop is dependent, the loop should not be executed in parallel.

2.4.2 TLS in IBM BG/Q

BG/Q is the latest IBM supercomputer in the BlueGene series (after BG/L and BG/P) that has hardware support for TLS and Transactional Memory (TM). The initial objective of BG/Q was to include TLS support but later TM support was also added to it. BG/Q requires hardware support for TLS, working in collaboration with a speculative run-time. The point of coherence for TLS in BG/Q is the L2 cache. Each different version of a memory address can be stored in a different way of the L2 cache. When a write occurs for a speculative thread, the L2 allocates a new way in the corresponding set for the write. A value stored by a speculative write is private to the thread and is not made visible to other threads. The value is made visible to other threads when a thread commits and is discarded upon a thread squashing. In addition, the L2 directory records, for each memory access, whether it is read or written, and whether it is speculative. For speculative accesses, the hardware also tracks the thread that has read or written the line by recording the speculation ID used by the thread to activate speculation. This enables the hardware to detect conflicts among threads and also between speculative and non-speculative thread.

The buffering of speculative states in the L2 requires co-operation from components of the memory subsystem that are closer to the pipeline than the L2, namely, the L1 cache. BG/Q can support two speculative execution modes for proper interaction between the L1, the L1 prefetcher (L1P) and the L2, each with a different performance consideration. The main difference between the two modes is in how the L1 cache keeps a speculative threads writes invisible to the other three hardware threads sharing the same L1.

- **Short-running (SR) mode (via L1-bypass):** In this mode, when a TLS thread stores a speculative value, the core evicts the line from the L1. Subsequent loads from the same thread have to retrieve the value from that point on from L2. If the L2 stores multiple values for the same address, the L2 returns the thread-specific data along with a flag that instructs the core

to place the data into the register of the requesting thread, but to not store the line in the L1 cache. In addition, for any speculative load served from the L1, the L2 is notified of the load via an L1 notification. The notification from L1 to L2 goes out through the store queue.

- **Long-running (LR) mode (via TLB aliasing)**: In this mode, speculative states can be kept in the L1 cache. The L1 cache can store up to 5 versions, 4 speculative ones for each of the 4 SMT threads, and one non-speculative. To achieve this, the software creates an illusion of versioned address space via Translation Lookaside Buffer (TLB) aliasing. For each memory reference by a speculative thread, some bits of the physical address in the TLB are used to create an aliased physical address by the Memory Management Unit (MMU). Therefore, the same virtual address may be translated to 4 different physical addresses for each of the 4 TLS threads at the L1 level. However, as the load or store exits the core, the bits in the physical address that are used to create the alias illusion are masked out because the L2 maintains the multi-version through the bookkeeping of speculation ID. In this mode the L1 cache is invalidated upon entering the speculative thread because there is no L1 notification to the L2 on an L1 hit. The invalidation of the L1 cache makes all first TLS accesses to a memory location visible to the L2 as an L1 load miss.

  These two modes are designed to exploit different locality patterns. By default, an application runs under the long running (LR) mode [49]. The support for SR mode is given for Transactional memory in BG/Q, but not for TLS yet. The main drawback of the short-running mode is that it nullifies the benefit of the L1 cache for read-after-write access patterns within a speculative thread. But it is well suited for short-running speculative threads with few memory accesses.

  The long-running mode, on the other hand, preserves temporal and spatial locality within a speculative thread, but, by invalidating L1 at the start of a thread, prevents reuse between code that run before entering the thread and code that run within the thread or after the thread ends. Thus, this mode is best suited for long-running threads with plenty of intra-thread locality.

The IBM *xlc* compiler has been modified to give speculation support in BG/Q and it is called *bgxlc_r*. The _r extension at the end generates thread safe code. The following pragma is used to speculatively execute a loop in parallel:

#pragma speculative for

By default, if this pragma is used, iterations of the loop are divided into *chunks* of size *ceiling(number_of_iterations/number_of_threads)*. Each thread is assigned a separate chunk.

2.5 LLVM

The profiling part of the speculative parallelizing framework, *SpecEval*, has been implemented in the LLVM compiler infrastructure [27]. The modular structure of LLVM makes it very easy to work with. This section describes the different tools and passes that are useful for loop dependence analysis.

1. **The *opt* tool**

   The *opt* tool of *llvm* is useful to run different optimization-related passes. There can be *analysis* passes that performs analysis on IR constructs such as: loops, CFG and Call Graphs but that do not modify the IR. The *transformation* passes perform the transformations for optimization.

2. **Useful Passes**

   Following are some passes that are used in implementation of the SpecEval framework [27].

   - The **-scev** (ScalarEvolution) is an analysis pass that can be used to analyze and categorize scalar expressions in loops. This pass specializes in recognizing general induction variables, representing them with the abstract and opaque SCEV class. Given this analysis, trip counts of loops and other important properties can be obtained.

   - The **-mem2reg** (Promote Memory to Register) is a transformation pass that converts non-SSA (static single assignment) form of LLVM IR into SSA form, raising loads and stores to stack-allocated values to 'registers' (SSA values). Many of LLVM optimization passes operate on the code in SSA form.

   - The **-loops** (Natural Loop Information) is an analysis pass that is used to identify natural loops and determine the loop depth of various nodes of the CFG.

   - The **-loop-simplify** is a transformation pass that performs several transformations on natural loops to change them into a simpler form that makes subsequent analyses and transformations simpler and more effective. Loop pre-header insertion guarantees that there is a single, non-critical entry edge from outside of the loop to the loop header. Loop exit-block insertion guarantees that all exit blocks from the loop (blocks that are outside of the loop and that have predecessors inside of the loop) only have predecessors from inside of the loop (and are thus dominated by the loop header).

   - The **-memdep** pass is an analysis pass that performs memory-dependence analysis. This

pass is based on the alias-analysis pass. The following types of alias analysis are available in LLVM.

- The **-basicaa** pass is an aggressive local analysis that knows many important facts such as:
  - Distinct globals, stack allocations, and heap allocations can never alias.
  - Globals, stack allocations, and heap allocations never alias the null pointer.
  - Different fields of a structure do not alias.
  - Indexes into arrays with statically differing subscripts cannot alias.
  - Many common standard C library functions never access memory or only read memory.
  - Function calls can not modify or reference stack allocations if those allocations never escape from the function that allocates them (a common case for automatic arrays).
- The **-globalsmodref-aa** pass implements a simple context-sensitive mod/ref and alias analysis for internal global variables that do not have their address taken. If a global does not have its address taken, the pass knows that no pointers alias the global. This pass also keeps track of functions that it knows never access memory or never read memory. The real power of this pass is that it provides context-sensitive mod/ref information for call instructions. This information allows the optimizer to know that calls to a function do not clobber or read the value of the global, allowing loads and stores to be eliminated.
- The **-steens-aa** pass implements a variation on the well-known 'Steensgaards algorithm' for interprocedural alias analysis [42]. The Steensgaards algorithm is a unification-based, flow-insensitive, context-insensitive, and field-insensitive alias analysis that is also very scalable (effectively linear time).
- The **-scev-aa** pass implements AliasAnalysis queries by translating them into ScalarEvolution queries. This translation gives this pass a more complete understanding of pointer instructions and loop induction variables than other alias analyses have.

The dependence analysis reports *may*, *must* and *no* dependence information. Instrumentation is performed for the *may* dependent instructions reported by the dependence analysis. The next chapter describes the details of a combined-profiling framework, implemented in LLVM, which is used to find dependence behaviour variation of loops based on inputs in different applications.

Chapter 3

Combined Data-Dependence Profiling

3.1 Introduction

Though previous work mentions that there is little variability in a loop's dependence behaviour based on inputs [45, 14, 25], there has not been an extensive study to find out if the claim is true. Also there have not been any proposals on how to combine data-dependence profiles obtained from multiple profiling runs of programs that will have loops with varied dependence behaviour. This chapter describes how to use CP [4] in the case of data-dependence profiling. Combined data-dependence profiles are built incrementally thus removing the necessity to store multiple profiles. The combined-profile file stores as much dependence information as is necessary to perform a cost analysis to find speculation candidates. This chapter describes a new combined data-dependence profiling framework implemented in the LLVM compiler infrastructure. Results show that for the 57 benchmarks that were widely used before in TLS literature, a loop's dependence behaviour *does not* change for different inputs.

A discussion on the effect of varied percentages of dependence on TLS performance of an application (incremental algorithm to build a convex 2D hull) is also included. As more applications of this kind are discovered, the proposed combined data-dependence profiling methodology will become useful.

3.2 The Combined Data-dependence Profiling Framework

The combined data-dependence profiling framework is implemented in the LLVM compiler. Within the framework, first a *transformation* pass, *profile-dependence*, is run on the source-code to instrument the bitcode (IR) for profiling. Another *analysis* pass, *printDbgInfo* is run to store the loop IDs and their corresponding file names and line numbers in a *log* file. Both of these passes need the bitcode to be produced with the debug information (-g option of clang). The *profile-dependence* pass uses three built-in LLVM passes: *mem2reg*, *loops* and *loop-simplify* (these passes have been

described in detail in Section 2.5 of Chapter 2). The instrumented bitcode is run with different inputs to produce a combined-profile file. This profile file can be consulted to find the dependence behaviour of speculation-candidate loops.

The following sections give more details about the different components of the framework.

3.3 The Instrumentation Pass

The instrumentation pass (called *profile-dependence*) performs IR instrumentation by adding calls to library-functions used for profiling. This section describes the implementation details of the pass.

3.3.1 Choosing Loops for TLS

The pass is written as a ModulePass in LLVM. For a given bitcode, the pass iterates through all the modules. For a given module, the pass iterates over the functions of the module and for a given function, the pass iterates over the basic blocks in the function.

In LLVM, the first basic block of a *for* loop is named as *for.condX* where X is any integer (depends on the number of loops in the program). Thus, while iterating through the basic blocks of the function, if the pass finds a basic block with *for.cond* in it's name, a variable *loop_count* is incremented.

The *loop_count* variable is used as a loop-identifier. The variable is of type unsigned integer and is also used to insert a unique global variable - *iteration_count* per loop that is used in identifying the current iteration ID of the loop (because they will be needed later to calculate the dependence distance). *Iteration_count* is cleared at the exit block of the loop so that the counter can be reused for another execution of the same loop.

For being a speculation candidate, the loop has to have the following properties -

- Branching in or out of structured block and parallel/work-sharing loop is not allowed. Therefore, if the loop has multiple exit blocks, the loop is not a speculation candidate. Multiple exit blocks exist when there are one or more jumps (*e.g. goto*-s) in the loop body.
- The loop should be countable. An example of non-countable loop is given in Figure 3.2.

```
for (; *s != '\0'; s++)
    *s = sre_toupper((int) *s);
```

Figure 3.1: Example of a non-countable loop from SPEC2006 *hmmer* benchmark.

- If the loop has non-intrinsic function calls (functions that can't be analyzed and replaced by an *intrinsic function* known to the compiler) inside it's body, then the loop is not safe to parallelize. But this constraint does not block a loop from executing speculatively in parallel. The

compiler can either relax the constraint or conservatively not parallelize loops with function calls (see Section 5.4.3 of Chapter 5).

Different built-in functions of LLVM are used to check the special characteristics of the loop.

- **Early exit condition of the Loop**
  LLVM has a function, *getUniqueExitBlock*, that returns either the unique exit block of a loop or returns null if the loop has multiple exit blocks.

- **Countable Loops**
  The *getSmallConstantTripCount*() function from the *ScalarEvolution* pass is used to identify countable loops. The function returns 0 if the trip count is unknown or not constant.

- **Function Calls**
  For checking if the loop body has function calls, the *Call* instructions inside the loop body are identified. If the call is to a C-library function, the body of the function will not be included in the bit code. If the *call* instruction is not a C-library call, the body will be there in the bitcode. The *empty*() function of LLVM's *Function* class is used to make this distinction. If *empty*() returns true, the *call* is to a C-library function, otherwise not.
  After filtering out *calls* to C-library functions, a check is necessary to find out whether the function may access memory. If the function may access memory, the loop is *not* parallelized because the side effects of the function call may alter the dependence behaviour of the loop (see Section 5.4.2 of Chapter 5) and an inter-procedural dependence analysis is necessary in that case.

3.3.2 The Algorithm Used for Instrumentation

The instrumentation pass is implemented as a *ModulePass* in LLVM. Algorithm 1 is used for the instrumentation. Basically the algorithm iterates over all the modules in the program and next over all the functions in the module and over all the basic blocks in the function.

When the first basic block of the function is found, a void pointer is allocated in the bitcode that is used to store the different memory accesses by the *load* and *store* instructions that *may* be dependent. This pointer is also used as an argument to the profiling functions [Lines 4-6].

If the basic block's name contains '*for.cond*', the pass detects a loop. The assumption is that no previous optimization(s) have been applied to the bitcode and that all the loops' basic blocks appear in the same order in which the loops are found in the source code. If a basic block with '*for.cond*' in its name is found, the $loopCount$ variable is incremented and two new global variables are also created in the IR [Lines 8-13].

- **loop_id** is an unique loop ID for the *for* loop.

- **iteration_id** is required to identify the iteration number of the memory access. This variable is incremented at the entry point of the loop body every time during the loop execution. Also *iteration_id* is cleared at the exit from the loop so that the variable can be reused for the next execution of the *for* loop.

Therefore, two global variables per loop are used. Only loops that have the properties mentioned above are instrumented. The function *candidateLoop*() checks for these properties. The function takes the help of *LoopInfo* and *ScalarEvolution* passes of LLVM.

After creation of the global variables, the exit block of the loop is identified using the *getUnique-ExitBlock*() function. This function always returns a unique basic block because a check is run to verify that the loop has a single exit block. Next, two instructions are inserted in the basic block. The first instruction clears the *iteration_id* so that the variable can be reused in the next execution. A *call* instruction to the *analyseAndWrite*() function is inserted. This function analyses the stored accesses to find the dependence pattern and to gather the profile information [Lines 14-17].

Next the loads and stores are identified and calls to the helper functions for profiling are inserted accordingly. There can be two types of basic blocks where the *loads* and *stores* can reside. Either *for.body* that comes just after the loop-conditional checking and *for.end* that is the last basic block of the loop.[1] The loads and stores in these basic blocks are checked to see whether they are reported as *may* dependent by the static analysis. If they are, their memory accesses are stored in the *void* pointer created before and *call* instructions to the profiling functions are inserted. This function takes the memory address accessed, loop ID, iteration ID, and the type of access (load/store) as parameters. More on this function is described later [Lines 19 -29].

If the basic block is the loop body (*for.body*), instructions to increment the *iteration_id* for the specific loop are also inserted [Lines 30-33].

The collected profile is written to the profile file only once - during the exit from the program. There are two ways for a program to exit -

- Return from *main*
- Calling the *exit*() function from any functions.

Therefore, the instructions are checked in the basic block to see whether there is a *return* instruction from *main* or whether there are *call* instructions to the *exit*() function. If such instructions are found, then the function that accesses the profile file and writes the collected profile information from memory to file is called [Lines 35 - 49].

[1] The second condition happens when there is loop nesting and there are instructions in the outer loop after the execution of the inner loop.

Algorithm 1 The new instrumentation algorithm used for profiling.

```
1:  for all mod in Modules do
2:    for all func in mod do
3:      for all basic_block in func do
4:        if basic_block.isFirstBasicBlockOfFunction() then
5:          allocateVoidPointer()
6:        end if
7:        if basic_block.getName().contains(for.cond) then
8:          loopCount++
9:          loop = getLoopFor(basic_block)
10:         if loop then
11:           if candidateLoop(loop) then
12:             createGlobal(loop_id)
13:             createGlobal(iteration_id)
14:             endBB = getUniqueExitBlock(loop)
15:             createInstruction(makeIterationCountZero)
16:             createInstruction(callAnalyseAndWrite)
17:             insertInstructions()
18:
19:             if basic_block.isLoopBody()||basic_block.isLoopEnd() then
20:               for all instruction in basic_block do
21:                 if isMayLoad()||isMayStore() then
22:                   createInstruction(getAddress(void_p))
23:                   createInstruction(getLoopID())
24:                   createInstruction(getIterationID())
25:                   createInstruction(call_common())
26:                   insertInstructions()
27:                 end if
28:               end for
29:             end if
30:             if basic_block.isLoopBody() then
31:               createInstruction(incrementIterationCount)
32:               insertInstructions()
33:             end if
34:           end if
35:           if func == main then
36:             for all inst in basic_block do
37:               if inst.isReturn()||inst.isCallToExit() then
38:                 createInstruction(callWriteToFile())
39:                 insertInstructions()
40:               end if
41:             end for
42:           else
43:             for all inst in basic_block do
44:               if inst.isCallToExit() then
45:                 createInstruction(callWriteToFile())
46:                 insertInstructions()
47:               end if
48:             end for
49:           end if
50:         end if
51:       end if
52:     end for
53:   end for
54: end for
```

3.4 The Profiling Library

In this section, the functions and the data structures used in the newly written profiling library are described. LLVM passes inserts calls to functions in this library to prepare the IR for profiling.

3.4.1 Helper Functions for Profiling

For each load-store instruction pair that is reported as *may*-dependent by the static dependence analysis, a function call is inserted that behaves differently for loads and stores. The following functions are the most important functions in the library.
*void common(void * x,int count,int loop, char type)* is the main function that is used for dependence detection. The working algorithm for this function is given in Algorithm 2.

The dependence behaviour is estimated from a portion of the loop's iteration space, the *loop sample*. The number of iterations to be considered for estimation is tunable by the macro *MAX_ITERATIONS*. When this function is called for the first time, it performs a one-time initialization of the data structures, that remains fixed for the whole profiling run. These data structures are stored in static memory because the use of dynamic memory for this data results in significant memory fragmentation.

Until the number of iterations as defined in the *loop sample* size is reached, whenever a *store* in encountered, it is added to a set, by the function *add_store*. Whenever a *load* is encountered, it is checked against the already gathered *store* instructions because we only care about loads that depend on previous stores (RAW dependences). Checking of write-after-read (WAR) and write-after-write(WAW) dependences is not necessary because in BG/Q TLS, threads always commit in order. If the access addresses are the same, a *dependence pair* is found. A dependence pair is defined as a tuple (i_1, i_2) for a given loop where i_1 is the iteration ID of the store and i_2 is the iteration ID of the load. Note, i_1 is always less than i_2. All the dependence calculated for an execution of a given loop are temporarily stored to find the dependence patterns.

When the loop enters an iteration that just exceeds the *loop sample* size; the collection of dependence pairs is analyzed to find the dependence behaviour for that execution of the loop. There can be three types of dependences for a given loop.

- **No Dependence** If no dependence pairs are found for all the executions of the loop, the loop can safely be executed in parallel.
- **Irregular Dependence** If the dependence does not follow any pattern (the dependence pairs have varied dependence distances), the loop is difficult to parallelize.
- **Strided Dependence** If the loop has a dependence with a stride value of 'n' (dependence is repeated for each 'n' iterations), every 'n' iterations of the loop can be executed in parallel, given the loop is doing significant work in those 'n' iterations to overcome the overhead from TLS.

The function $check\_stride()$ checks if there was a strided dependence pattern. The function uses the state machine described in Figure 3.3 to find strided dependence. First the function calculates a dependence distance (distance between the iteration id of the write and that of read) for a given dependence pair.

Next the function checks for dependence pairs with the same dependence distance in the higher iteration numbers. For example, if a dependence pair $< 1, 3 >$ is found (data written in iteration 1 is read by iteration 3), the function checks for dependence pairs $< 2, 4 >$, $< 3, 5 >$ etc. It is worth mentioning that if there are strided dependences with parallel executions in the middle (strided dependence with uniform breaks), still the function reports an irregular dependence because forming chunks for speculative execution of that loop is non-trivial. Every time a dependence pair is encountered, a counter is incremented. If the counter value reaches a certain threshold (the threshold ID set as 5), a strided dependence is reported and the dependence information is kept in memory until the program exists (when it is written to disk). Whenever a matching dependence pair is not found, the dependence becomes irregular. Also, to reduce the search space, already visited dependence pairs are not searched again.

After analyzing and storing the dependence pattern in the memory, all the data structures used

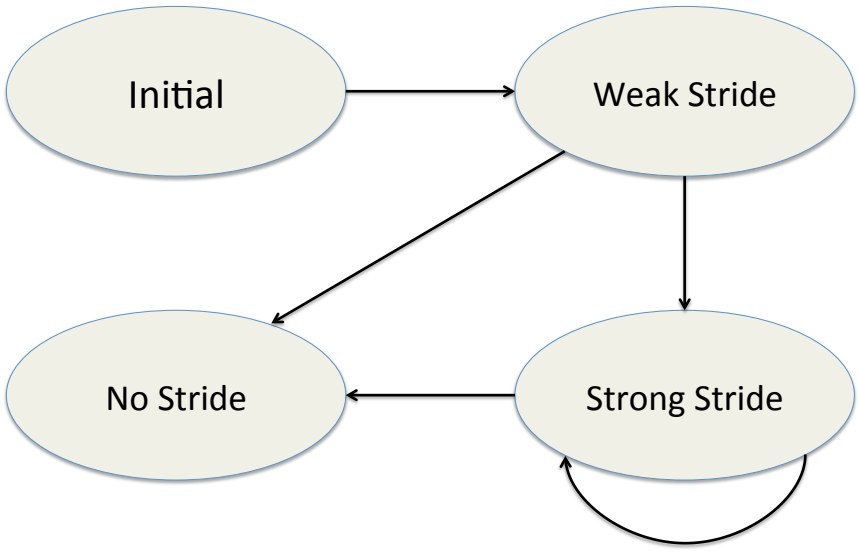

Figure 3.2: The state machine used for detecting strided dependence.

in that particular execution of the loop are cleared so that they can be reused in the next run. This is achieved by just clearing some counters, but not resetting the whole data structures. Also the *flag*, indicating that the dependence analysis is done, is set in the *setAnalyzedFlag()* function. The flag is checked in the *analyse_and_write()* function to find out whether the dependence analysis of the loop is already performed. For loops smaller that MAX_ITERATIONS, this flag enables the dependence analysis in the *analyse_and_write()* function.

int analyse_and_write(int loop) is used for the loops whose iteration count is smaller than *MAX_ITERATIONS*. This function is inserted at the exit block of the loop. If the *flag* indicating the dependence analysis is not set, then the function calculates the dependence behaviour of the loop. Otherwise, the function just resets the *analyzed* flag so that the function *common()* can work in the next loop execution. This function is useful for calculating dependence information for loops smaller than *MAX_ITERATIONS*.

All the dependence information is kept in data structures temporarily until they are written to a file before the program terminates. If there is already a profile file, the *write_to_file()* function updates the file using the information from the current run. Otherwise, the function creates a new profile file *llvmprof.out*. A call to this function is only inserted before the *return* instruction from *main* and before the *call* instruction to the *exit* function.

Algorithm 2 The new algorithm for the $common()$ function, the common() function finds dependence pairs and also characterizes the dependence as one of: *strided*, *irregular* or *no-dependence.*

```
1:  if iteration_id <= MAX_ITERATIONS then
2:    if data structures uninitialized then
3:      initializeDS()
4:      setInitializedFlag()
5:    end if
6:    if load instruction then
7:      check_dependence()
8:    else
9:      add_store()
10:   end if
11: else
12:   if iteration_id <= MAX_ITERATIONS + 1 then
13:     if dep_pair_count == 0 then
14:       loop parallel
15:     else
16:       check_stride()
17:       if stride found then
18:         strided dependence
19:       else
20:         irregular dependence
21:       end if
22:     end if
23:     set_analyzed_flag()
24:     reset_DS()
25:   end if
26: end if
```

3.4.2 Data Structures Used by Helper Functions of Profiling

The various data structures that are used by the functions in the profiling library are *static* arrays of structures. Linked list is not a good choice here for the following reasons:

- Memory fragmentation is created due to the use of malloc().
- Linked lists need an extra storage for storing the pointers to next and previous elements.

The different structures used by the profiling library functions are as follows:

- **Memory Accesses**

  Only *stores* are needed to be gathered because the *loads* are checked on fly. A *void* pointer is used to store memory addresses of any type. A static array of the structure is used to gather information about *stores*. The MAX_LOOPS parameter is tunable and it gives the upper bound of the number of loops in a program. The value is set to 5000. $MAX\_ACCESSES$ gives the upper bound of the number of stores that can occur for an execution of the *loop sample*. The value is set to 2000.

- **Dependence Pair**

  The *DependencePair* structure has three members. $write$ and $read$ are used to store the iteration id of the store and load, respectively. $checked$ is a flag that is used during the stride calculation to avoid redundant computation. A similar array of structures is used to store the dependence pairs. The maximum number of dependence pairs stored is chosen as 2000 following the results from experimentation.

- **Profile Information**

  In this structure, $loop\_id$ stores the unique ID of the loop. $parallel$ and $irregular$ variables are used to store the number of parallel executions and the number of executions with irregular dependence. The bins of the histogram are used to store the stride values for dependence. A static array of structures is used to store the profile information in the memory.

- **Miscellaneous Data Structures**

  Apart from the main data structure mentioned above, the following arrays are also used -

  1. To keep flags of already analyzed loops.
  2. To keep track of the number of memory access count and dependence pair count. These counters reduce the costly traversal of the arrays of data structures to prepare them for the next execution of the loop.
  3. To store the discovered stride value, if applicable.

The total size of the data structures remains fixed because they are initialized once and reused throughout the execution of the program. For 5000 loops, 2000 accesses and 2000 dependence pairs, the memory overhead comes to 200 MB.

3.5 The Profile file

The profile file produced by the framework stores the dependence information for a number of executions for each loop. The combined-profile file tries to capture as much dependence information as is necessary to perform a cost analysis to find speculation candidates.

As there can be three types of dependence behaviour as discussed before, for each loop, the combined-profile file stores the unique loop ID and the number of independent (parallel) and irregularly dependent executions. The file also stores the different stride values in a histogram with five bins. Table 3.1 shows a sample profile-file. Normally it's a sparse matrix because dependence behaviour does not vary much with the inputs. A consolidated representation is also possible to make the file smaller. But for the applications considered, the file size never exceeds 20KB.

Table 3.1: A sample combined-profile file. Detailed information about the dependence behaviour of a loop is kept to perform a cost analysis.

loop_id	parallel_executions	irregular_executions	first_bin	second_bin	third_bin	fourth_bin	fifth_bin
120	250	0	0	0	0	0	0
121	2	2	0	0	0	0	0
240	12	5	0	2340	0	0	0
309	340	12	0	0	21	0	0
569	2	2	0	1	0	0	0

3.6 Variability of a loop's dependence behaviour based on inputs

Using the combined dependence-profiling framework described above, 57 different benchmarks from the SPEC2006 [1], PolyBench/C [38], BioBenchmark [35] and NAS benchmark [34] suite are studied with different inputs (Table 3.2). Results show that *there was not a single loop that has the specific properties for being a TLS candidate as described in Section 3.3.1 and whose dependence behaviour varies with different inputs*. This finding allows the use of a simple heuristic for selection of speculation candidates based on profiling. If the loop is dependent, the loop is not speculated. If the loop is found to be independent, it is speculatively parallelized. This invariability in the dependence behaviour also allows to use a single input for profiling and finding out TLS candidate loops in most applications.

Table 3.2: The benchmarks observed to find loops with varied dependence behaviour.

Benchmark Suite	Benchmark Name
SPEC2006	lbm h264ref hmmer mcf sjeng sphinx3 bzip2
Continued on next page	

Table 3.2 – continued from previous page

Benchmark Suite	Benchmark Name
	gobmk
	milc
	namd
PolyBench/C	2mm
	3mm
	gemm
	gramschmidt
	jacobi
	lu
	seidel
	cholesky
	dynprog
	fdtd_2d
Biobench	mummer
	protdist
	protpar
	dnapars
	dnamove
	dnapenny
	dnacomp
	dnainvar
	dynprog
	dnaml
	dnaml2
	dnamlk
	dnamlk2
	dnadist
	dollop
	dolmove
	dolpenny
	restml
	restml2
	seqboot
	fitch
	kitsch
	neighbor
	gendist
	tigr
	clustalw
	hmmer
NAS	BT
	CG
	DC
	EP
	FT
	IS
	LU
	MG
	SP
	UA

3.7 An Application with a Loop with Varied Dependence Behaviour for Different Inputs

Though the loop's dependence behaviour does not change with respect to inputs for most applications, a description of an application (2D-Hull) that has loops with varied dependence behaviour across different executions is included in this section. The existence of any other application that has the loops with similar behaviour, is not known yet.

Table 3.3: Effects of different input sets when computing the Convex Hull.

Input	Percentage of Dependence Violations
Kuzmin	0.001
Square	0.005
Disc	0.035
Circle	10.400

2D-Hull: The randomized incremental algorithm that builds the Convex Hull of a two-dimensional set of points is used as an application. This algorithm, called 2D-Hull (due to Clarkson *et al.* [10]), computes the convex hull (smallest enclosing polygon) of a set of points in the plane. The input to Clarkson's algorithm is a set of (x, y) point coordinates. The algorithm starts with the triangle composed by the first three points and adds points in an incremental way. If the point lies inside the current solution, it is discarded. Otherwise, the new convex hull is computed. Any change to the solution found so far generates a dependence violation, because other successor threads may have used the old enclosing polygon to process the points assigned to them.

The probability of a dependence violation in the 2D-Hull algorithm depends on the shape of the input set. For example, if N points are distributed uniformly on a disk, the *i*-th iteration will have a dependence with probability in $\phi(\sqrt{i}/i)$. If the points lie uniformly on a square, the probability of a dependence will be in $\phi(log(i)/i)$.

Four Different input sets are considered for the performance study of this application. Each of the input contains 10-million points.

- **Kuzmin**: Kuzmin is an input set that follows a Gauss-Kuzmin distribution, where the density of points is higher around the center of the distribution space. This input set leads to very few dependence violations because points far from the center are very scarce.
- **Square**: It is an uniform distribution of points inside a square.
- **Disc**: This input is an uniform distributions of points inside a disc. The Square input set leads to an enclosing polygon with fewer edges than the Disc input set, thus generating fewer dependence violations.
- **Circle**: The circle input set distributes all the points around a circle, leading to a huge number of dependence violations.

3.7.1 Experimental Results

This section describes the effect of different inputs on the dependence behaviour of the loop of the Convex hull application and the effect of speculative execution of the loop on the execution time. Table 3.3 gives the percentage of loop iterations that have a dependence on previous iterations for

Convex Hull. The percentage is highest for *circle* and lowest for *kuzmin*. This dependence violation percentage also affects the execution time of the application. Figure 3.3 shows the percentage change in speedup for applying TLS along with the auto-SIMDizer and auto-OpenMP parallelizer of the bgxlc_r compiler.

Irrespective of the different probabilities of dependences for different inputs, the loop in the Convex hull application is executed speculatively in parallel to find out how the different percentages of dependence violation impacts the performance of the application. The loop is not parallelized by the existing parallelization techniques of the *bgxlc_r* compiler (IBM's *xlc* compiler modified for BG/Q).

The high percentage of dependence violations for the *circle* input set makes the loop non-beneficial for speculative execution. While for the *kuzmin* input, executing the loop speculatively improves performance 21% fold over the auto-SIMDized code. None of the loops are automatically parallelized by the SIMD or OpenMP parallelizer because the compiler can not prove the absence of dependence at compile time.

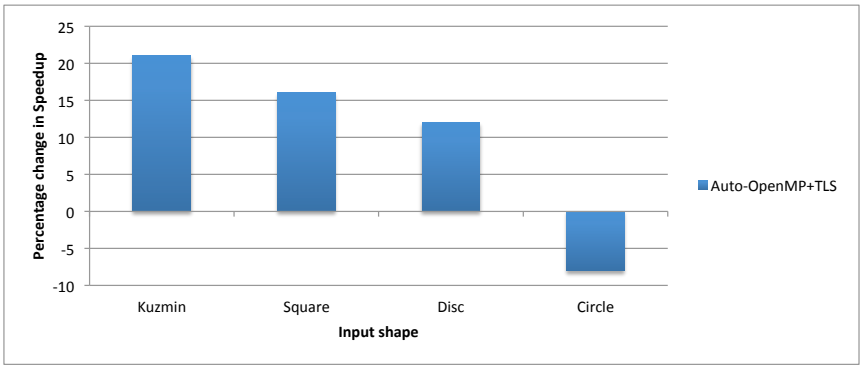

Figure 3.3: Percentage change in speedup for applying TLS as compared to the auto-SIMDized code for the Convex hull application.

3.8 Conclusion

In conclusion, this chapter shows the implementation of a combined data-dependence profiling framework that is used to find out variability in a loop's dependence behaviour in 57 different benchmarks. Even though no variation in the dependence behavior based on input is found in the loops of the wide range of benchmarks studied, the methodology described in this chapter for combined dependence profiling is a new contribution and may become useful in the future if dependence behavior variation is identified in a given application domain. One such application is the incremental algorithm that builds the Convex Hull, as described in Section 3.7. As more applications like these are discovered, a cost analysis can be performed to select speculatively parallel loops based on the probability of dependence at run-time.

In the next chapter, an automatic speculative parallelization framework, SpecEval, is described that uses the dependence analyzer of *Polly* (the polyhedral optimizer in LLVM) to find *may dependences* inside loops. Data-dependence profiling is shown to be important because allowing speculative execution of loops with *may dependences* without checking whether the dependences materialize during runtime, results in a slowdown. The impact of thread creation overhead is shown to be important because allowing speculative execution of cold loops also results in a slowdown. When these two kind of loops (cold loops and loops with *may dependences* that materialize during run-time) are filtered out, the slowdown is tackled. From here on, data-dependence profiling indicates a single-input profiling where the inputs to the profiling run and test run are different.

Chapter 4

SpecEval: A framework for automatic speculative parallelization of loops

4.1 Introduction

This chapter presents SpecEval, an automatic speculative parallelization framework that can parallelize loops automatically using some heuristic for TLS. SpecEval uses the dependence analyzer of *Polly* [19] to find *may dependences*. Two different heuristics are used to find speculation candidate loops. The first heuristic allows loops with only *may-dependences* to run speculatively in parallel, irrespective of whether the dependences materialize during run-time. This heuristic shows the importance of data-dependence profiling to reduce the mispeculation overhead in TLS. In the first heuristic, the coverage of loops is also not considered while selecting speculation candidates. This relaxation gives an estimation of the high thread creation overhead in case of loops with low coverage. The second heuristic filters out *cold* loops and, using dependence-profile information, loops with actual run-time dependences. A performance evaluation of the speculative code generated by SpecEval over the sequential version at the lowest optimization level (-O0) of the bgxlc_r for the SPEC2006 and PolyBench/C benchmarks is presented. A performance evaluation of TLS applied with the automatic OpenMP parallelizer of *Polly* is also included.

4.2 Speculative Parallelization using the polyhedral dependence analyzer of LLVM

The two heuristics used by the SpecEval to find speculation candidate loops are as follows:

4.2.1 Heuristic 1

In the first heuristic, for being a speculation candidate a *SCoP* (loop) should have only *may dependences* (SCoPs are described in Section 2.1.1 of Chapter 2) . The goal of this heuristic is to relax the

constraint for OpenMP parallelization (OpenMP does not parallelize loops with *may dependences*) and find more parallelization candidates. The hope is that the *may dependences* will not materialize at run-time, thus resulting in speedup.

4.2.2 Heuristic 2

Heuristic 1 allows loops with *may dependences* to execute in parallel. But there can be two cases where the overhead of speculation can degrade the performance. In heuristic 2, the two criteria for filtering are based on these two different overheads. The first criteria considers the overhead from mispeculation and recovery while the second criteria considers the overhead from storing the program state during thread creation that is necessary for the system to roll back to a consistent state in case of mispeculation.

1. In the loops where *Polly* reports only *may dependences*, the memory accesses are profiled for a training run with one input. If the training run shows that the *may dependences* are materializing into *actual dependences* at run-time, the loop is not parallelized.

2. If the execution time of the loop is less than some threshold *coverage* (percentage of whole program execution time), the loop is not speculatively parallelized. Because in this case, the overhead from thread creation can negate the performance of the program. This threshold is set as 1.2% of the total execution time because results show that allowing the speculative execution of smaller (colder) loops leads to slowdown.

4.3 Description of SpecEval

SpecEval is implemented using the LLVM [27] and bgxlc_r (IBM's *xlc* modified for BG/Q) compiler. SpecEval uses LLVM passes to find *may dependences* inside loops, for profiling and for generating debug information so that source-code instrumentation can be performed. SpecEval uses bgxlc_r to produce executable code from the instrumented source code because TLS support is not yet implemented in LLVM. SpecEval inserts a speculation pragma before a loop that is has decided to speculatively execute in parallel. SpecEval uses two different dependence analysis passes to find *may dependences*: 1) the polyhedral dependence analysis; and 2) the built in dependence analyzer of LLVM. Polyhedral dependence analysis is part of the LLVM project and it is called *Polly* [19]. This chapter includes results using Polly's dependence analyzer while the next chapter presents results using the built-in dependence analyzer of LLVM.

Figure 4.1 describes the SpecEval framework. There are three phases of SpecEval.

1. Collection of profile and debug information:
   This phase contains several steps. In the first step, the source code is compiled to Intermediate Representation (IR) with the `-g` option of the LLVM compiler so that debug information to

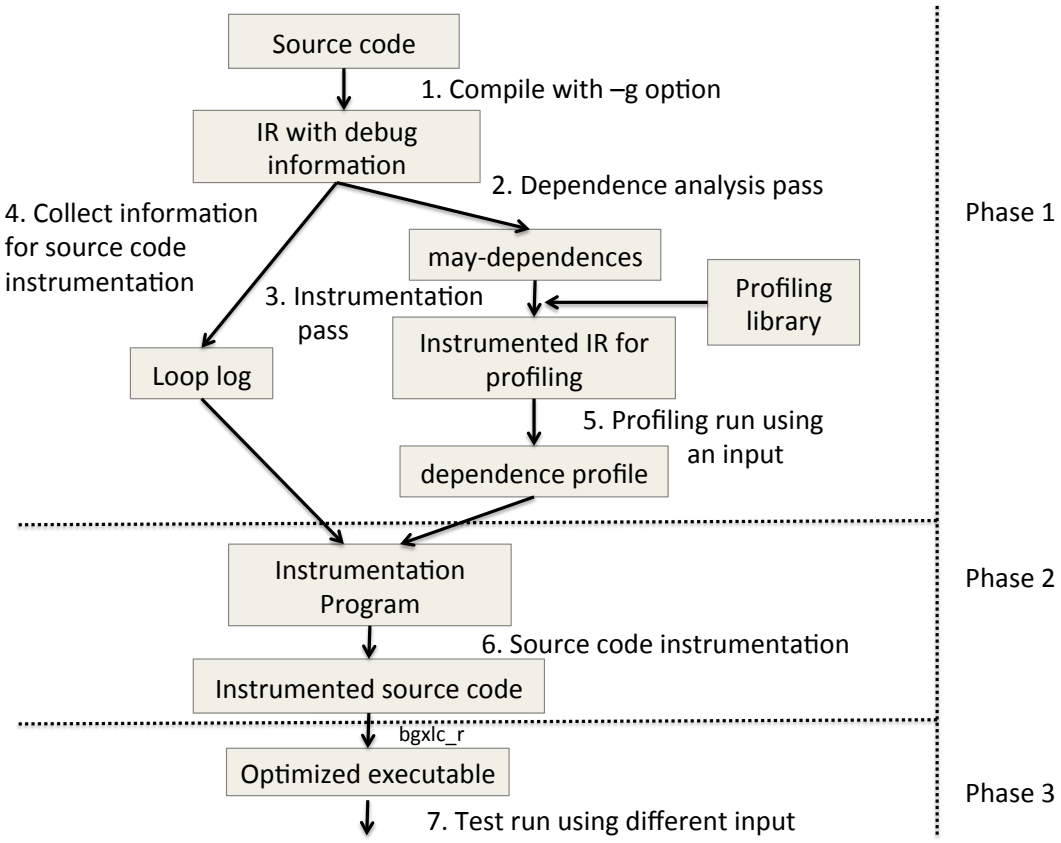

Figure 4.1: The speculative parallelization framework-SpecEval.

map executable code to source code can be collected. This debug information is necessary to determine the source-code file name and line number where each speculative loop appears so that SpecEval can insert the speculation pragma. Step 2 involves running a dependence-analysis pass to find *may dependences*. The dependence-analysis pass can be either from Polly or from LLVM. In step 3, a newly written instrumentation pass inserts calls in the IR to prepare itself for profiling. The functions reside in a newly written profiling library. Step 4 collects the debug information about the loops in the program. Polly reports loops as SCoPs and the SCoPs are tracked in the source code using their basic-block information. For the experiments with LLVM's built-in dependence analyzer, a new pass is written that collects source-code file name and line number for all the loops in the program. This debug information is stored in a file indicated as *Loop log* in the figure. Lastly, in step 5, the instrumented IR produced at step 3 is run using an input to collect the dependence profile.

2. Source-code instrumentation:

   A newly written C program takes the *Loop log* and the dependence-profile file as inputs and inserts speculative pragmas in the source code before *for* loops that are found to be as speculation candidates following different heuristics.

3. Test run:

   The instrumented source code produced in the previous phase is compiled with the *bgxlc_r* compiler to produce an executable that can be run in the BG/Q.[1] *bgxlc* is the IBM *xlc* compiler specific for the BG/Q machine. A different input is used in the *profiling* run than the input used for the *test* run.

4.4 Experimental Evaluation

Experimentation is performed with two different sets of benchmarks - SPEC2006 benchmarks [1] and the PolyBench/C benchmarks [38]. SPEC2006 is chosen because it is used widely in TLS research. All the SPEC2006 benchmarks are not included in the report because they do not run successfully on BG/Q. PolyBench/C benchmarks are chosen because (1) they are suitable benchmarks for polyhedral analysis and (2) they are used in TLS literature before. Table 4.1 shows the hardware specifications of a BG/Q chip.

The SPEC2006 benchmarks are run with the *ref, train and test* input and the PolyBench/C benchmarks are run using varied problem size. For calculating the speedups, each benchmark is run a total 60 times for a given input and the average running time from the 60 runs were taken. For runs with profiling, the input for the profiling run is different from the input for the test run. The 95% confidence interval is not shown in the bar chart because it is too small to be significant. A one sample student's t-test performed on the 60 execution times for different benchmarks showed that they have p-values in the range 0.12-0.34 (a p-value less than or equal to 0.05 means the variation in statistically significant).

Instrumentation in the source code is done automatically following the information file generated by SpecEval. The loop must be countable at compile time to be a speculation candidate.

The baseline for comparison is the sequential version of the benchmarks compiled with the -O0 optimization level of the *bgxlc_r* compiler. The lowest optimization level ensures that the optimization of hot loops (-qhot) and automatic SIMDization of the sequential code (-qsimd=auto) is turned off. To see the performance impact of TLS when applied with an auto-OpenMP parallelizer, the automatic OpenMP code generated by *Polly* is used. Polly inserts calls to OpenMP run-time functions to parallelize independent SCoPs.

[1] The _r option generates thread-safe code.

Table 4.1: Hardware specifications of a BlueGene/Q chip.

#Processors	17(16 User and 1 service PowerPC)
Multithreading	4-way Multithreaded
Clock	1.6GHz
L1 I/D Cache	16KB/16KB
Peak Performance	204.8 GFLOPS 55W
RAM	16 GB DDR3
Multiversioned Cache	Support for Transactional Memory and Speculative Execution
L2 Cache	Centrally shared, 32 MB
Chip-to-chip networking	5D Torus topology + external link

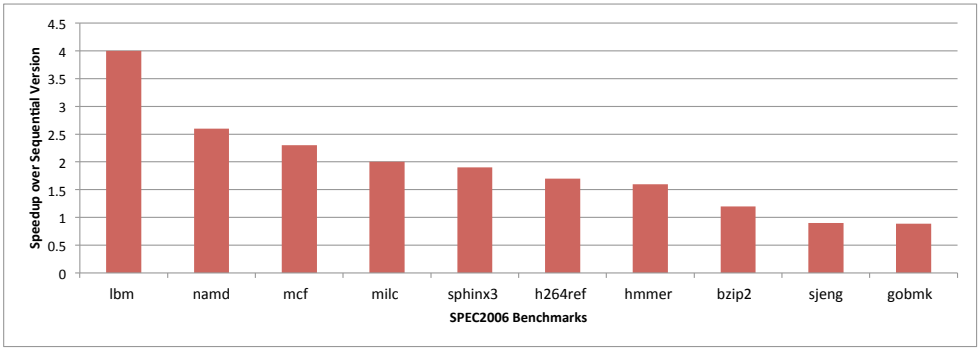

Figure 4.2: Speedup of the TLS version of SPEC2006 benchmarks over the optimized sequential version for 4 threads. *gobmk* suffers a slow down because of the presence of many loops with small iteration count. *sjeng* experiences a slow down because the *may* dependences materialize during run-time.

4.5 Results

This section describes the results for the experimentation with *Polly*. First the effect of heuristic 1 on the SPEC2006 and the PolyBench/C benchmarks is described. Then a performance evaluation of TLS when applied with Polly's auto-OpenMP parallelizer is presented. The effect of heuristic 2 on the two sets of benchmarks is also shown. Results show that data-dependence profiling is necessary because in benchmarks such as *sjeng*, *may dependences* materialize during run-time that causes a slowdown. The results also show that speculatively parallelizing loops with poor *coverage* results in slowdown for benchmarks such as: *fdtd-2d*, *jacobi*, *lu*, *seidel*, *cholesky*, *dynprog*, *gramschmidt* and *gobmk*. Applying heuristic 2 to filter out loops with actual dependences and *cold* loops (less than 1.2% coverage) tackles the slowdown.

4.5.1 Heuristic 1

Spec2006 Benchmarks

In Figure 4.2, most of the SPEC2006 benchmarks achieve a speedup over the optimized sequential version. *lbm* contains loops with no inter-thread data-dependences, but these dependences are not

statically provable by the compiler and that's why they are not parallelized by OpenMP. This benchmark can be greatly benefited by the speculative execution and obtains the highest speedup because *may dependences* do not materialize at run-time.

gobmk has many loops with small iteration counts. Small loops are not good candidates for speculative execution because the thread creation overhead negates the impact of parallel execution and we get a slowdown. These loops are later filtered out by heuristic 2.

sjeng contains loops that are reported as speculation candidates by heuristic 1 due to the occurrence of *may dependences*. But these dependences materialize during run-time and the overhead from mispeculation prevents *sjeng* from achieving speedup. These loops are not suited for speculative execution and they are eliminated by data-dependence profiling.

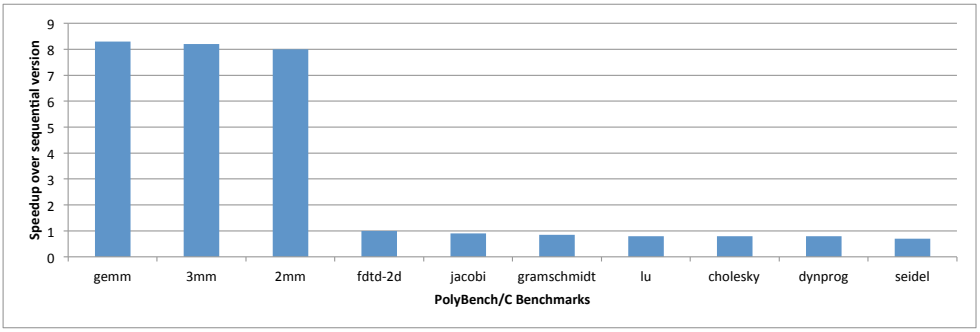

Figure 4.3: Speed up of the TLS version of PolyBench/C benchmarks over the optimized sequential version for 4 threads.

PolyBench/C Benchmarks

Polybenchmarks were run with different sizes of input for calculating the speedup. The speedups reported in Figure 4.3 indicate that the PolyBench/C programs can be divided into two classes according to the effectiveness of thread-level speculation. Class 1 contains programs such as: *2mm*, *3mm* and *gemm* that achieve speed up due to TLS and Class 2, containing *gramschmidt*, *jacobi*, *lu*, *cholesky*, *dynprog* and *seidel* experience a slow down. In the Class 2 PolyBench/C programs the loops parallelized have small iteration count as well as *coverage* (the loops mainly initialize arrays).[2] Therefore the overhead for thread creation in the speculative execution negates the performance achieved from the parallel execution of these loops. Table 4.2 shows the coverage of the speculatively parallelized loops from the SPEC2006 and PolyBench/C benchmarks. For *seidel* the coverage is only 0.04% and therefore the loops are not good speculation candidates. The cold loops are eliminated by heuristic 2.

[2]These results confirm the finding of Kim et al. [25]

Table 4.2: Number of loops parallelized by auto-OpenMP parallelizer of Polly and speculative parallelization using heuristic 1. Heuristic 1 allows loops with *may-dependences* and no *must-dependences* to be executed in parallel. Coverage data is omitted where there is no speculative loop discovered.

Suite	Benchmark	Total	OpenMP	Speculative	Coverage of Speculative Loops
SPEC2006	lbm	23	4	4	97
	namd	619	9	20	92
	mcf	52	9	4	60
	milc	421	7	20	68
	sphinx3	609	11	2	91
	h264ref	1870	179	45	79
	hmmer	851	105	30	80
	bzip2	232	4	2	35
	sjeng	254	9	3	12
	gobmk	1265	0	1	.7
PolyBench/C	gemm	13	3	4	98
	3mm	27	10	-	-
	2mm	20	7	-	-
	fdtd_2d	14	2	0	-
	jacobi	9	3	3	2
	gramschmidt	10	3	2	.9
	lu	8	3	2	2
	cholesky	9	0	1	.9
	dynprog	9	7	2	1
	seidel	7	4	1	0.04

4.5.2 Effects of Applying TLS along with Auto-OpenMP Parallelizer

Heuristic 1 relaxes the constraint of OpenMP parallelization and allows loops with *may dependences* and no *must dependences* to speculatively run in parallel. Figures 4.4 and 4.5 show the percentage change in speedup when TLS is applied with the automatic OpenMP parallelizer of Polly for the SPEC2006 and PolyBench/C benchmarks respectively. As seen in Figure 4.4, *lbm*, *namd*, *milc*, *h264ref* and *bzip2* have a small performance improvement due to TLS but *sjeng* and *gobmk* perform worse due to TLS for the reasons mentioned before. From Figure 4.5, for *2mm* and *3mm*, TLS does not give any performance improvement because no new speculation candidate loops are discovered by SpecEval. In the slowdown benchmarks, the loops parallelized by heuristic 1 have a small iteration count and they take very small portion of the benchmark execution time. Therefore executing them speculatively causes the overhead from TLS to negate the gain from parallel execution. The slowdown in these benchmarks is the motivation for applying heuristic 2.

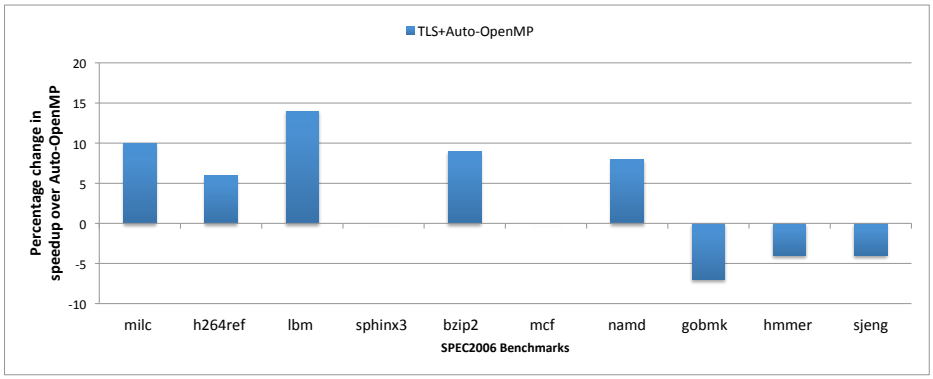

Figure 4.4: Percentage change in speedup after applying TLS over auto-OpenMP parallelizer of Polly for SPEC2006 benchmarks using 4 threads.

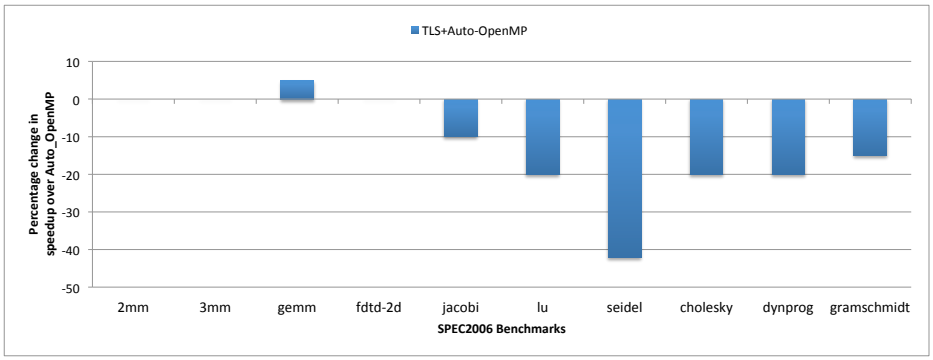

Figure 4.5: Percentage change in speedup after applying TLS over auto-OpenMP parallelizer of Polly for PolyBench/C benchmarks using 4 threads.

4.5.3 Heuristic 2

Profiling and Filtering hot loops

To tackle the slowdowns, the reported *may dependences* are profiled to see whether the dependences materialize during run-time. As it is previously shown (see Section 3.6 of previous chapter) that loop's dependence behaviour does not change according to inputs, a single input data-dependence profile is used where the input for profiling run is different from the input for the test run. Loops that do not take significant amount of the program execution time (*cold* loops) are not speculatively parallelized because they have huge thread creation overhead.

The results shown in Figure 4.6 indicate that the benchmarks that were experiencing slowdown performs equal or better when heuristic 2 is used. TLS versions of *gramschmidt* and *sjeng* perform better than the OpenMP parallelized versions because the overhead from parallelizing *cold* loops goes away.

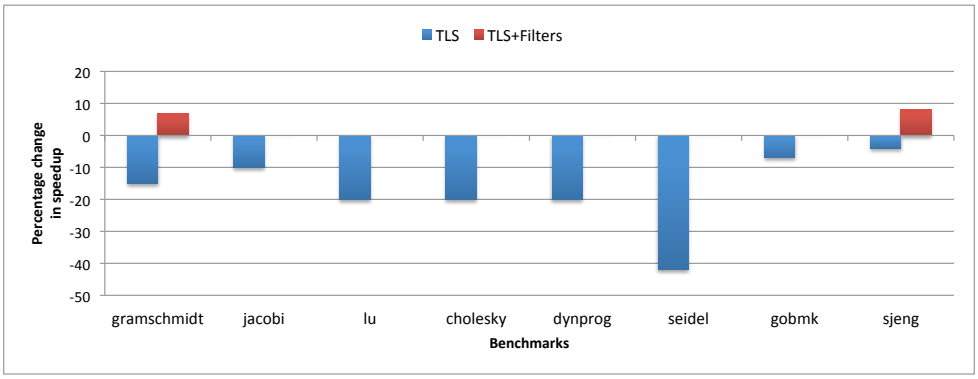

Figure 4.6: Percentage change in speedup after applying heuristic 1 and heuristic 2 to the benchmarks that were experiencing slowdown. Heuristic 2 performs equal or better than auto-OpenMP for *all* cases after filtering cold loops and loops with a run-time dependence.

4.5.4 Scalability

For measuring the scalability of the speculatively parallelized benchmarks, the speculative code is run with increasing number of threads. Figure 4.7 shows the scalability of the TLS version of the SPEC2006 benchmarks. *lbm* and *milc* scale upto 16 threads. *mcf* is a benchmarks that is benefited from L1 prefetching effect. As seen from Figure 4.8, there is very little performance variation with increasing number of threads for the PolyBench/C benchmarks.

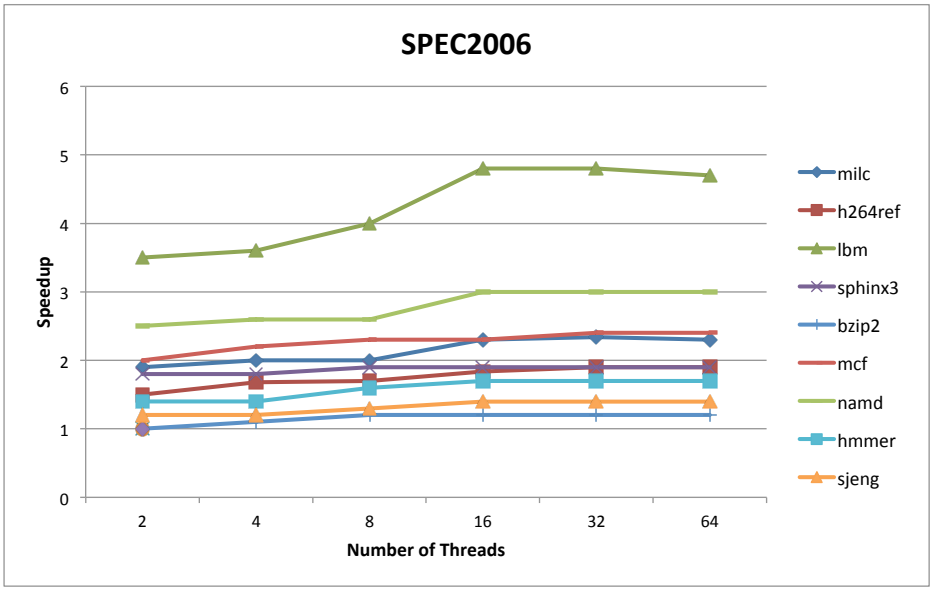

Figure 4.7: Scalability of speculatively parallelized versions of the SPEC2006 benchmarks with Polly.

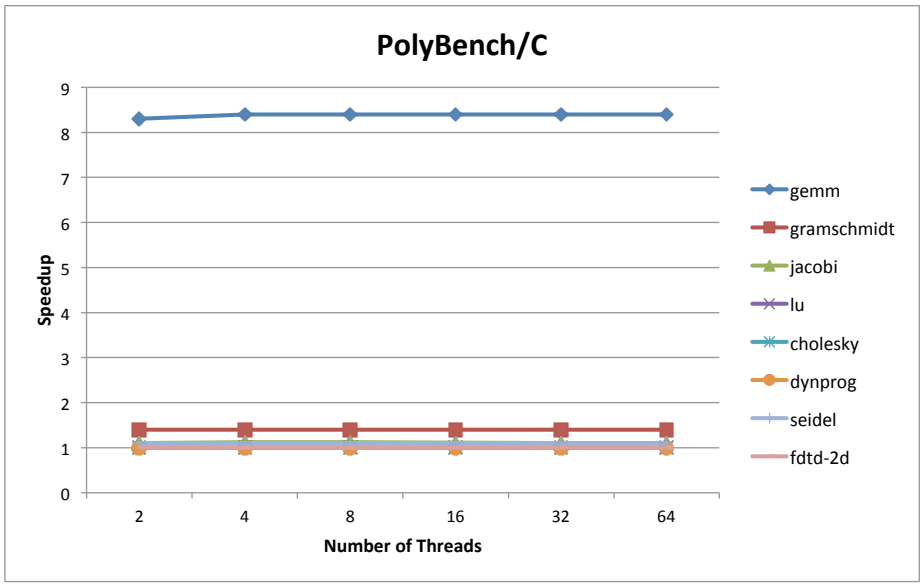

Figure 4.8: Scalability of speculatively parallelized versions of the PolyBench/C benchmarks with Polly.

4.6 Conclusion

In this chapter, an automatic speculative parallelization framework SpecEval is described that parallelizes loops that are reported to have *may dependences* by the static dependence analyzer of *Polly*. Profiling is shown to be important in selecting TLS candidate loops because allowing speculative execution of loops with *may dependences* that materialize during run-time causes slowdown in *sjeng*. It is also shown that speculative execution of *cold* loops causes slowdown due to thread creation overhead.. When the *cold* loops and loops with *may dependences* that materialize during run-time are filtered out, the slowdown is tackled.

In the next chapter, SpecEval takes the dependence analysis results from the new and sophisticated built-in static dependence analyzer of LLVM instead of the dependence analyzer from *Polly* . The next chapter also presents a performance evaluation when TLS is applied along with the traditional parallelization techniques such as: (1) SIMDization and (2) OpenMP parallelization by the *bgxlc_r* compiler. The performance evaluation in the next chapter uses the sequential code compiled with the highest optimization level (-O5) as the baseline. A study on the various factors that impact TLS performance are included.

Chapter 5

Study of Speculative Parallelization at Higher Optimization Levels of the Compiler

5.1 Introduction

The higher optimization levels of the compiler are used by programmers to improve the performance of sequential code. To obtain maximum performance improvement, programmers typically use the highest possible optimization level of a compiler (*e.g.* -O5 for IBM's *xlc* compiler, -O3 for *gcc*).

To extract parallelism from programs, different automatic parallelization techniques, such as SIMDization and OpenMP parallelization, are used. This chapter explores the effect of applying speculative parallelism to the SPEC2006 and PolyBench/C benchmarks, with the sequential code optimized at the highest level (-O5) of the bgxlc_r compiler taken as baseline. This chapter studies the performance of the benchmarks when TLS is applied along with the existing parallelization techniques (auto-SIMDization and auto-OpenMP parallelization in the bgxlc_r compiler). Data-dependence profiling has already been found to be useful for TLS (see Section 4.5.3 of previous chapter). A single input data-dependence profile is used to identify loops that are candidate for speculation because dependence behaviour of loops does not change with inputs for the SPEC2006 and PolyBench/C benchmarks (see Section 3.6 of Chapter 3). The dependence analysis pass of LLVM is used to find *may dependences* inside loops (instead of Polly's dependence analysis pass as described in the previous chapter). Loops with *may dependences* are profiled to discover whether the *may dependences* materialize at run-time.

The performance evaluation of the SPEC2006 and PolyBench/C benchmarks uses three parallel versions of the code: (1) code generated by the existing automatic SIMD parallelizer of *bgxlc_r*; (2) OpenMP parallelized along with SIMDized code generated by the *bgxlc_r*; and (3) OpenMP + SIMDized code by *bgxlc_r* along with the speculatively parallelized code generated by SpecEval (described in Section 4.3 of the previous chapter).

The speedups obtained from an *oracle* version of the benchmarks are also shown for compari-

son. The oracle version is obtained by applying TLS incrementally to candidate loops. If applying TLS to a candidate loop degrades the performance improvement that was obtained so far, that loop is rejected from TLS by the Oracle. In this way, the oracle version offers the best possible TLS candidate loop set.

Following the results described in Section 4.2.2 of previous chapter, the threshold of coverage (percentage of whole program execution time) for selecting speculation candidate loops is kept as 1.2%.

Results show that for benchmarks such as *namd*, *mcf* and *lbm*, TLS moderately improves the execution time while, for some benchmarks, TLS results in a performance degradation. Factors that impact the TLS performance include: (1) number of loops speculatively parallelized and their coverage; (2) increase in L1 cache misses due to Long-running (LR) mode (see Section 2.4.2 in Chapter 2) in BG/Q (*sjeng*, *cholesky*, *dynprog*), (3) mispeculation due to dependences resulted from function calls (*sjeng*, *h264ref*) and (4) increase in dynamic instruction path length (*jacobi*, *seidel*).

There is mispeculation overhead if a function call inside the loop body introduces new dependences at run-time. This situation did not arise for the experiments with Polly because Polly does not consider a loop as SCoP if there is a function call inside the loop body. But those loops are included for consideration by the new dependence analyzer of LLVM. If there is a function call inside the loop body, the memory accesses inside the called function are not profiled to find a dependence. If all the memory accesses other than the ones inside the *callee* are found to be independent, the loop is considered to be a speculation candidate. Even if the function call introduces dependences inside the loop during run-time, those dependences are not discovered through off-line profiling. The reason for this relaxation is the complexity of performing an inter-procedural dependence analysis with the help of off-line profiling. For performing such analysis, the calling context of the function has to be considered. If there is a loop within the callee, an inter-loop dependence analysis is necessary to find out cross-iteration dependences. Due to this complexity of profiling, a more conservative compiler does not consider a loop as a candidate for speculation if there is a function call inside the loop body and the function is not side-effect free. To minimize the overhead due to dependences introduced at run-time from function calls, a filter is applied to prevent the speculative execution of loops with function calls. There is one caveat of the filtering criteria: it prevents parallel execution of loops where the function calls do not introduce dependence. But results show that loops where a function call does not introduce dependences either do not occur or are too cold to change the performance significantly in the benchmarks studied.

A scalability study of the speculatively parallelized versions of SPEC2006 and PolyBench/C benchmarks is also included.

5.2 Experimental Evaluation

This section evaluates the impact of applying profile-driven TLS to the SPEC2006 and PolyBench/C benchmarks, with the BG/Q machine as the hardware platform. The inputs used for the profiling (training) run and the test run are different. To calculate the speedup, an average execution time of 60 runs is considered. 95% confidence intervals for the measurement of speedups are not shown in the graphs because they are too small to be significant. A one sample student's t-test performed on the execution times show that the p-values are in the range between 0.15-0.33 (a p-value less than or equal to 0.05 means a significant variation).

5.3 Effect of applying TLS with AutoSIMD and AutoOpenMP parallelizer in the bgxlc_r

This set of experiments studies the effect of applying TLS, along with existing parallelization techniques in the SPEC2006 and PolyBench/C benchmarks, with the sequential code optimized at the highest optimization level of the *bgxlc_r* compiler as the baseline. Speedup is measured for three parallel versions of the code as mentioned before. By default the -O5 level turns on the automatic vectorization (SIMDization) and the various optimizations of hot loops (*e.g.* loop unrolling etc.).

The result from the dependence analysis pass of LLVM is used for profiling by SpecEval because LLVM's dependence analysis pass was improved by the time these experiments are performed. The SPEC2006 and PolyBench/C benchmarks are used for the experimental evaluation.

The following compiler options are used to generate the sequential and the three parallel versions of the code.

- **Optimized Sequential version (baseline):** bgxlc -O5 -qsimd=noauto -qnohot -qstrict -qprefetch
- **Automatic SIMDized version:** bgxlc -O5 -qsimd=auto -qhot -qstrict -qprefetch
- **Automatic OpenMP parallelized + SIMDized version:** bgxlc_r -O5 -qsimd=auto -qhot -qsmp=auto -qstrict -qprefetch
- **Speculative parallelized + OpenMP + SIMDized version:** bgxlc_r -O5 -qsimd=auto -qhot -qsmp=auto:speculative -qstrict -qprefetch

The qstrict compiler option is used to maintain the correct semantics of the program after higher-level optimizations because the higher-level optimizations may alter the semantics of the program. The -qprefetch option enables prefetching.

Figure 5.1 and figure 5.2 show the percentage change in speedup of the OpenMP and TLS parallel versions of the SPEC2006 and PolyBench/C benchmarks respectively over the auto-SIMDized code. The highest percentage change that can be obtained from an oracle version is also shown.

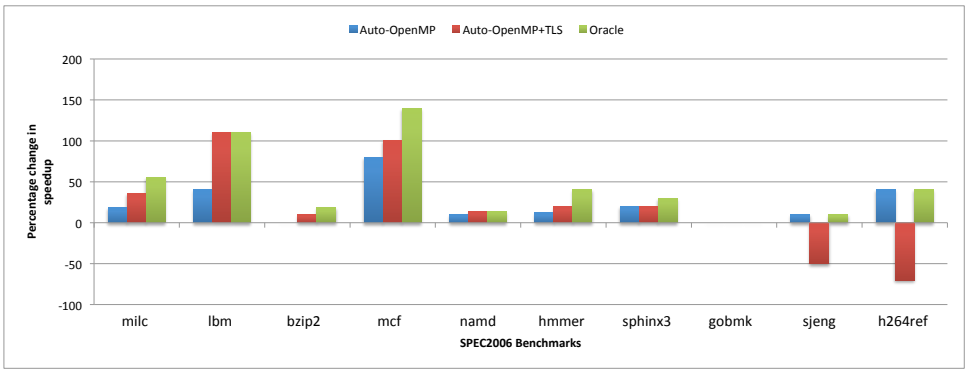

Figure 5.1: Percentage change in speedup obtained from different parallelization techniques for the SPEC2006 benchmarks over auto-SIMDized code. The autoOpenMP and autoOpenMP+TLS versions use 4 threads.

There are three classes of benchmarks - benchmarks that achieve some speedup with TLS (*milc*, *lbm*, *bzip2*, *mcf*, *namd*, *hmmer*), benchmarks where performance neither improves nor degrades (*sphinx3*, *gobmk*) and benchmarks that get a performance degradation with TLS (*h264ref*, *sjeng*). For the *mcf* benchmark, the oracle version is seen to have a significant performance improvement over the TLS version by SpecEval. Closer inspection revealed that there are loops that have poor coverage and mispeculation due to function calls. These non-beneficial loops are eliminated by the oracle version as they impact the TLS performance. These loops can be eliminated by SpecEval by increasing the coverage threshold and eliminating TLS execution of loops that have function calls.

As seen in Figure 5.2, PolyBench/C benchmarks also fall under these three classes. The TLS version of *gemm* gets a 3.3x speedup and of *lu* gives slightly better performance than the SIMDized and OpenMP parallelized versions while in *jacobi*, *cholesky* and *dynprog* TLS experiences a slow-down. Performance does not improve after applying TLS to *2mm*, *3mm*, *fdtd-2d*, *gramschmidt* and *seidel* benchmarks.

One of the factors that impacts TLS performance is the number of parallelized loops and their *coverage*. Coverage is defined as the percentage of the loop's execution time with respect to the whole program execution time. There is limited performance improvement after applying TLS to loops that are not 'hot' enough due to the speculative thread creation overhead. *bzip2*, *sjeng* are benchmarks that contain speculative loops with poor coverage (Table 5.1).

Increase in L1 cache misses due to LR mode in TLS is another factor that limits the TLS performance. In BG/Q, for LR mode, the L1 cache is flushed before entering any speculative region thus affecting regions with data locality [21]. Benchmarks such as *cholesky* and *dynprog* suffer from the flush-effect of TLS (Table 5.3). The *Short-running* (SR) mode is more suitable for the speculative execution of these benchmarks but the SR mode for TLS is not yet available in BG/Q.

Events, such as saving of register context before entering a speculative region and obtaining a speculative ID, account for the TLS overhead that is reflected in the increase in instruction path

length. *jacobi* and *seidel* are two benchmarks that experience a huge path-length increase (Table 5.4) thus limiting/degrading their performance. The rest of the section explains in details the different sources of BG/Q TLS overhead.

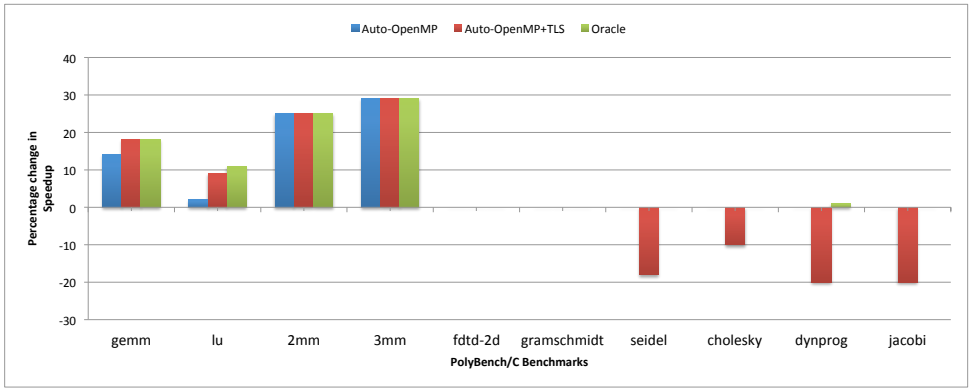

Figure 5.2: Percentage change in speedup obtained from different parallelization techniques for the SPEC2006 benchmarks over auto-SIMDized code. The autoOpenMP and autoOpenMP+TLS versions use 4 threads.

5.4 Impact of Different Factors on TLS Performance

To investigate the factors that impact the TLS performance in BG/Q, the following set of experiments is performed. The factors found to have impact on TLS performance are:

- The number and coverage (percentage of whole program execution time) of speculatively parallelized loops . If the loops have poor coverage, the thread creation overhead limits TLS performance.
- The mispeculation overhead due to introduction of run-time dependence by function calls. These dependences are not easily detectable using dependence profiling because the function call may be recursive and may contain other loops. The mispeculation overhead is measured in terms of the number of threads squashed during speculative execution.
- The L1 cache miss rate increase due to the LR mode.
- Increase in the length of dynamic path due to speculative execution of loops with low coverage.

5.4.1 Number of loops parallelized and their coverage

Table 5.1 shows the total number of loops in each benchmark, the number of parallelized loops by the three different parallelization techniques- automatic OpenMP parallelized, automatic SIMDized and speculatively parallelized, and the coverage of speculatively parallelized loops in the SPEC2006

and PolyBench/C benchmarks. *Polly*'s dependence analyzer only considers loops that are SCoPs. To be a SCoP, the loop has to have the properties described in Section 2.1.1 of Chapter 2. On the other hand, when the dependence-analysis result from the loop-dependence analyzer of LLVM is used to find *may-dependent* memory accesses, non-SCoPs are also profiled. As a result more loops are reported to be candidates for TLS based on the speculative candidate loop selection criteria as seen in a comparison of Table 4.2 and Table 5.1. The number of loops parallelized by the auto-OpenMP parallelizer is also greater than the number of loops parallelized by *Polly* for the same reason.

The interesting benchmark in Table 5.1 is *h264ref* because it has the highest number of spec-

Table 5.1: Number of loops parallelized by different parallel versions of the benchmarks and the coverage of speculatively parallelized loops. Coverage data are omitted where no speculation candidate loop is found.

Suite	Benchmarks	# Total loops	Speculative	OpenMP	SIMDized	Coverage of Speculative Loops
SPEC2006	lbm	23	5	4	0	98
	h264ref	1870	47	179	3	82
	hmmer	851	30	105	17	80
	mcf	52	12	9	0	65
	sjeng	254	16	9	0	32
	sphinx3	609	2	11	0	91
	bzip2	232	2	4	0	35
	gobmk	1265	0	0	0	-
	milc	421	22	7	2	33
	namd	619	25	9	7	92
PolyBench/C	2mm	20	-	7	3	-
	3mm	27	-	10	3	-
	gemm	13	4	3	4	40
	gramschmidt	10	0	3	0	-
	jacobi	9	2	3	0	3
	lu	8	5	3	1	45
	seidel	7	2	4	0	3
	cholesky	9	4	0	0	10
	dynprog	9	3	7	0	18
	fdtd_2d	14	3	2	0	20

ulatively parallelized loops with good coverage among all other benchmarks but still it experiences a slowdown due to speculative execution, as seen in Figure 5.1. Experiments reveal that function calls introduce dependences during run-time and those dependences are not detected with profiling as explained in Section 5.1. Therefore the mispeculation overhead limits the TLS performance of *h264ref*. *sjeng* experiences a slowdown for the same reason.

The performance improvement from TLS for *milc* is limited because it has a large number of speculative loops with low coverage. The poor coverage of loops introduces TLS thread creation overhead that causes a limited performance improvement. An ideal TLS candidate loop will have high coverage (hot loop) with no (low probability of) dependence violation. The number of loops

speculatively parallelized for *lbm* is small but they take a significant portion of the whole program execution time (98%). These hot loops of *lbm* are good speculation candidates and are examples of cases where TLS can be beneficial.

For PolyBench/C benchmarks *2mm* and *3mm*, no new speculation candidate loops are discovered because these benchmarks are matrix multiplication benchmarks that contain large parallelizable loops. These loops are parallelized by the automatic OpenMP parallelizer of *bgxlc_r*. For *gemm*, the speculatively parallel loops have good coverage that accounts for the 3.3x speedup from TLS. But for *cholesky*, *dynprog* and *fdtd-2d*, the poor coverage of loops results in slow-down.

gramschmidt does not have any speculative loops discovered because the *cold* loops (less than 1.2% coverage) that are parallelized by the *Polly* dependence analyzer (see Results in Chapter 4) are filtered out as those loops are already found to be non-beneficial for TLS.

5.4.2 Mispeculation Overhead

Squashing threads with dependence and re-executing the parallel section sequentially imposes overhead that results in performance degradation. For speculatively parallelizing loops with function calls, new dependences may be introduced that lead to dependence violation and thread squashing. The measurement of mispeculation overhead identifies loops inside benchmarks where function calls introduce actual dependences during run-time.

To measure the mispeculation overhead, the percentage of successfully committed threads are computed. The *se_print_stats* function from the *speculation.h* header file in BG/Q is used to collect various statistics for a speculative region including the number of successfully committed/non-committed threads. The percentage of speculative threads committed is computed using the following equation:

$$Percentage_{succ} = \frac{\#speculative}{(\#speculative + \#non\_speculative)} * 100 \tag{5.1}$$

The percentage of successfully committed threads gives an idea about the amount of wasted computation for the speculative loops. Ideally a benchmark that can benefit from TLS should have a high percentage of successful completion of speculative threads. Table 5.2 gives the percentage of speculative threads committed for the SPEC2006 and PolyBench/C benchmarks. The best TLS performance, in terms of successful thread completion, is in *lbm*. For *sjeng* and *h264ref* the percentage is much lower, giving an indication of a huge amount of wasted computation that causes their slowdown. Closer investigation of these benchmarks reveals that most of the loops speculatively parallelized contain function calls that introduces new dependences during run-time. Though *h264ref* has a high number of loops speculatively parallelized (Table 5.1), the presence of dependence resulted from the function calls inside the loops accounts for the slowdown. *mcf*, *hmmer*, *milc*, *namd* and *bzip2* also suffer from this phenomenon.

Among the PolyBench/C benchmarks, *gemm* and *lu* have a high percentage of speculative thread

completion that accounts for their speedup. For four of the benchmarks - *jacobi*, *seidel*, *cholesky* and *dynprog* there is a high percentage of thread completion but still these benchmarks experience slowdown. Experiments show that *cholesky* and *dynprog* suffer from L1 cache miss rate due to LR mode while *jacobi* and *seidel* suffer from huge increase in dynamic instruction path length due to the presence of loops that constitute most portion of the code but have poor coverage.

The following techniques can be used to overcome the mispeculation overhead due to function calls:

- The compiler can be conservative and may not allow loops with function calls inside them to be executed in parallel regardless of whether the function call changes the dependence behaviour of the loop. In this heuristic, we might still miss some opportunities where the function call is harmless.
- A more sophisticated inter-procedural dependence analysis technique is necessary that is able to tell whether the function call introduces new dependences during run-time.

Table 5.2: The percentage of speculative threads successfully committed (not rolled back) for the SPEC2006 and PolyBench/C benchmarks. The percentage gives the amount of wastage computation. Data is omitted for benchmarks that have no speculative loop.

Suite	Benchmark	Percentage of Speculative Committed
SPEC2006	lbm	94
	h264ref	12
	hmmer	79
	mcf	68
	sjeng	8
	sphinx3	29
	bzip2	78
	milc	79
	namd	80
PolyBench/C	gemm	89
	jacobi	78
	lu	89
	seidel	70
	cholesky	79
	dynprog	82
	fdtd_2d	68

The effect of using heuristic 1 on the SPEC2006 benchmarks is evaluated in the following section.

5.4.3 Filtering Loops with Function Calls that have Side Effects

As seen in Section 5.4.2, for the two benchmarks *h264ref* and *sjeng*, allowing speculative execution of loops with function calls introduces mispeculation overhead that results in slowdown. The function calls inside the loop bodies introduce dependences across iterations of the loop during

run-time. Static dependence analysis of the compiler is not able to detect these dependences. An inter-procedural dependence analysis with the help of profiling is difficult to perform.

In this section, the performance impact after preventing speculative execution of loops with function calls that may have side effects is explored.

As seen in Figure 5.3, preventing speculative execution of loops with function calls have a

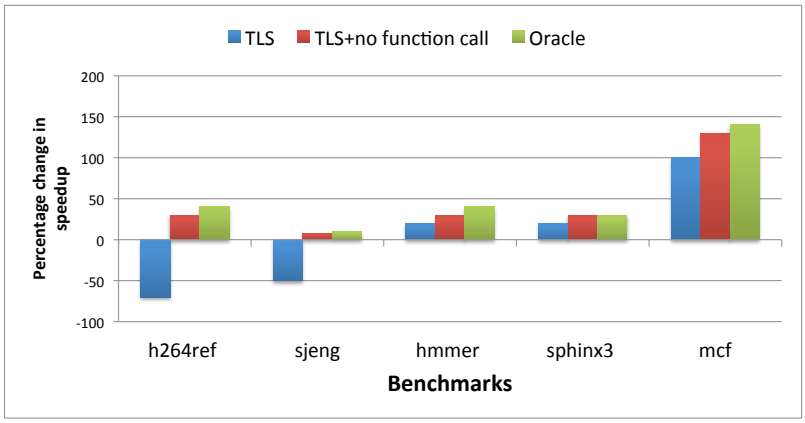

Figure 5.3: Percentage change in speedup of SPEC2006 benchmarks after filtering speculative execution of loops with function calls.

performance impact on the *mcf*, *h264ref* and *sjeng* benchmarks. The percentage of successfully committed threads jumps up from 12% to 96% for *h264ref* and from 8% to 97% for *sjeng*. The performance of *mcf* also goes very close to the performance of oracle. Performance *does not* degrade for any of these benchmarks indicating that there is not any loop in these benchmarks that is hot and has side-effect-free function calls.

The performance of the PolyBench/C benchmarks does not change when this heuristic is used (p-value of 0.54 in student's t-test). Mostly PolyBench/C benchmarks are kernel benchmarks and the loops inside them do not contain function calls.

5.4.4 L1 Cache Miss Rate

One of the most dominant BG/Q TLS run-time overhead is caused by the loss of L1 cache support due to the L1 flush and bypass needed for the bookkeeping of speculative state in L2. Though the L2 and L1P buffer load latencies are 13x and 5x higher than the L1 load latency, the L1 miss rate both for the sequential and parallel versions of the code gives an idea about the performance gain or loss for the benchmarks.

The Hardware Performance Monitor (HPM) library of BG/Q is used to collect the L1 miss statistics. Table 5.3 gives the L1 cache hit rate for the sequential version and the three parallel versions of the SPEC2006 and PolyBench/C benchmarks. The speculative execution of *sjeng* results in a high L1 miss rate. This high miss rate is the effect of flushing the L1 cache before entering

Table 5.3: L1 Cache hit rate (percentage) for the sequential and three parallel versions of the SPEC2006 and PolyBench/C benchmarks.

Suite	Benchmark	Sequential	SIMD	AutoOMP	Speculative
SPEC2006	lbm	95	94	94	93
	h264ref	96	95	95	94
	hmmer	98	97	97	95
	mcf	92	92	95	95
	sjeng	96	96	95	90
	sphinx3	96	96	95	95
	bzip2	95	95	95	97
	gobmk	97	97	97	97
	milc	95	97	97	98
	namd	96	98	97	98
PolyBench/C	2mm	98	98	99	99
	3mm	98	98	99	99
	gemm	98	96	98	98
	gramschmidt	97	97	97	97
	jacobi	97	97	97	97
	lu	96	96	95	96
	seidel	98	97	98	98
	cholesky	98	98	96	88
	dynprog	97	96	97	90
	fdtd_2d	98	98	98	98

the TLS region in the LR mode. Apart from the function calls that introduce data-dependences (Table 5.2), the high L1 miss rate affects the performance for *sjeng*.

Similar effect can be seen for the two PolyBench/C benchmarks - *cholesky* and *dynprog*. Though these two benchmarks have a high percentage of successful completion of speculative threads as seen in Table 5.2, the speculative execution of the selected loops affects the cache performance due to locality of data between threads. The cost of bringing the data again after flushing the cache accounts for the slowdown in these benchmarks.

For *jacobi* and *seidel* benchmarks, though the speculative execution of the loops result in better cache utilization, the benchmarks experience a slowdown. The reason for this slowdown is the increase in instruction path length. The two benchmarks *fdtd-2d* and *gobmk* do not experience any change in cache utilization for the three parallelization techniques (automatic OpenMP, SIMDization and speculative parallelization), because there are no parallelizable loops.

5.4.5 Instruction Path length increase

Automatic OpenMP and speculative parallelization inserts call to OpenMP and TLS run-time functions respectively in the parallelized loops. Code is also inserted for saving the consistent system state so that the system can be rolled back to a previous state in case of a dependence violation and thread squashing. The effect of TLS on code growth is shown in the results in Table 5.4.

The code growth for PolyBench/C benchmarks is higher than for SPEC2006 benchmarks. The code growth is due to loops constituting a major portion of the PolyBench/C benchmarks. Applying

Table 5.4: Percentages of dynamic instruction-path-length increase of the three parallel versions of the SPEC2006 and PolyBench/C benchmarks with respect to their sequential version.

Suite	Benchmark	SIMD	AutoOMP	Speculative
SPEC2006	lbm	.03 %	.25 %	26 %
	h264ref	.6 %	15 %	56 %
	hmmer	10 %	35 %	37 %
	mcf	0 %	12 %	23 %
	sjeng	0 %	0 %	45 %
	sphinx3	0 %	18 %	19 %
	bzip2	0 %	2 %	3 %
	gobmk	0 %	0 %	0 %
	milc	0.9 %	12 %	23 %
	namd	1 %	12 %	25 %
PolyBench/C	2mm	13 %	45 %	45 %
	3mm	13 %	46 %	46 %
	gemm	11%	45	45 %
	gramschmidt	0 %	46 %	46%
	jacobi	0 %	95	112 %
	lu	1 %	12 %	13 %
	seidel	0.02 %	98 %	123 %
	cholesky	0 %	0 %	99 %
	dynprog	0 %	0 %	75 %
	fdtd_2d	0 %	0 %	79 %

parallelization to the loops affects the code size more significantly. For SPEC2006 benchmarks the code growth is relatively smaller, the highest being for the speculative parallelization of the *h264ref* benchmark because a large number of loops are speculatively parallelized for this benchmark (Table 5.1).

As seen in Table 5.4, the two benchmarks *jacobi* and *seidel*, from the PolyBench/C benchmarks, experience a huge code growth that explains their slowdown. Benchmarks such as *cholesky*, *dynprog* and *fdtd_2d* also suffer code growth due to the presence of loops with poor coverage (see Table 5.1). Therefore these kind of loops (that suffer code growth) are not good candidate for speculative execution.

5.5 Scalability

As technology scales, the number of cores that can be integrated onto a processor increases. Thus, it is important to understand whether TLS can efficiently utilize all the available cores. In this section, the scalability of TLS performance is studied for the SPEC2006 and PolyBench/C benchmarks by comparing the speedup achieved using 2 to 64 threads. The results of this study are shown in Figure 5.4 and 5.5.

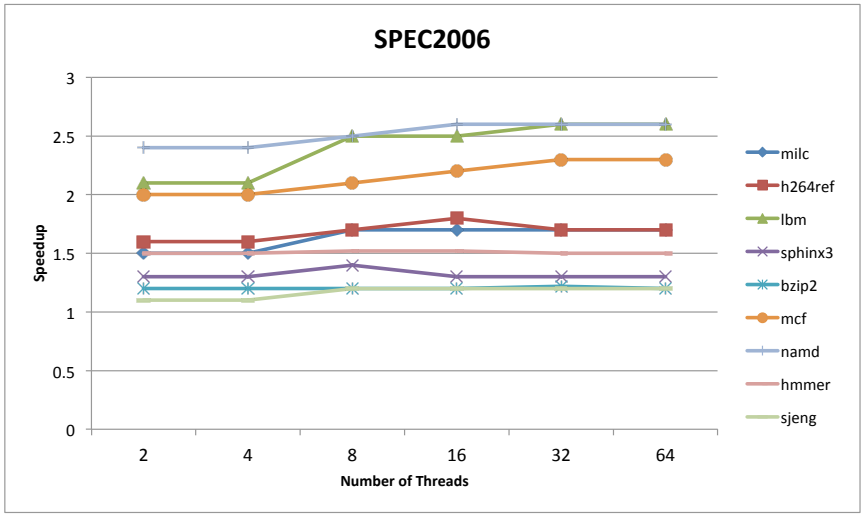

Figure 5.4: Scalability of speculatively parallelized versions of the SPEC2006 benchmarks.

5.5.1 SPEC2006 Benchmarks

As seen in Figure 5.4, *lbm*, *sphinx3*, *namd* and *h264ref* contain a number of loops with good coverage. Therefore, these benchmarks show some scalability with the increasing number of threads.

mcf is a benchmark that scales up to 32 threads due to cache prefetching [37]. The performance of *milc* scales up to 8 threads. For *hmmer* and *sjeng* the performance improvement for TLS is negligible in all configurations. While the reasons for the lack of scalability differ from benchmark to benchmark, it is obvious that the amount of parallelism is limited.

5.5.2 PolyBench/C Benchmarks

Figure 5.5 shows the scalability of the speculative versions of the PolyBench/C benchmarks. There is a little improvement for *gemm* but most of the other benchmarks experience no performance change when number of threads is increased.

5.5.3 A Discussion on the Use of Clauses with the Basic TLS Pragma

All the experiments discussed so far uses the basic TLS pragma available for TLS in BG/Q (see Section 2.4.2 of Chapter 2). But the *bgxlc_r* compiler offers many OpenMP-like clauses that can be used to optimize the performance of the speculative loop (see Appendix A to find the set of clauses available in BG/Q). These clauses offer more flexibility in the following two aspects:

- **Scoping of variables**: The clauses *default*, *shared*, *private*, *firstprivate* and *lastprivate* give the option to specify the scope of the variables used inside the loop.

- **Work Distribution**: The clauses *num_threads* and *schedule* give the option to change number

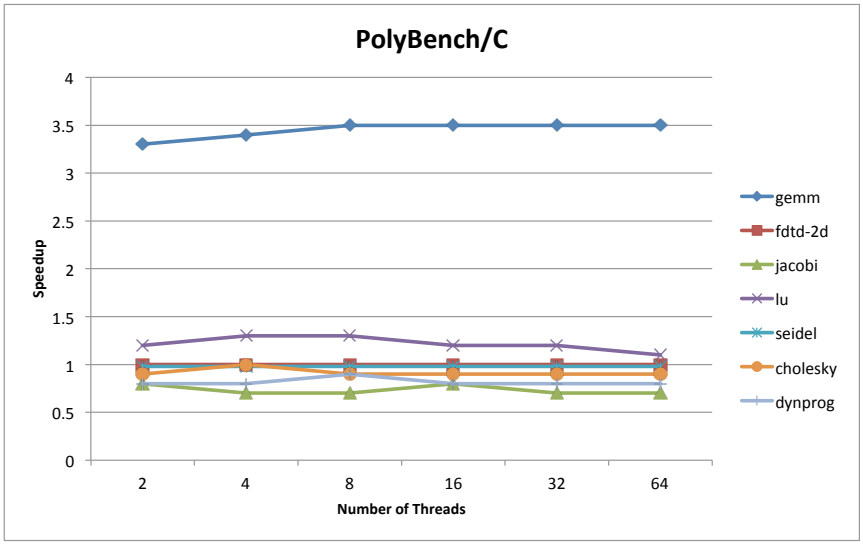

Figure 5.5: Scalability of speculatively parallelized versions of the PolyBench/C benchmarks.

of threads and distribution of work among threads.

This case study illustrates the use of specific clauses to the *lbm* and *h264ref* benchmarks. This choice of benchmarks is based on the fact that *lbm* has loops that are suitable for TLS execution and *h264ref* has the highest number of speculatively parallelized loops. Clauses are manually added to pragmas to study the performance impact of these clauses. This manual instrumentation opened the opportunity to speculatively parallelize more loops. For example, the loop in Figure 5.6 from the SPEC2006 *lbm* benchmark, is reported as dependent by the dependence analysis. Privatization of the variables ux, uy, uz, u2 and rho allows the execution of this loop in parallel because the compiler ensures that each thread has it's own local copy of the value specified as *private* and the values are duly forwarded among threads. The basic pragma is modified for the loop shown in Figure 5.6 as follows:

#pragma speculative for private (ux, uy, uz, u2, rho)

This study also investigates the impact of different work distribution strategies on the TLS performance for the speculatively parallelized loops for the two benchmarks. The performance evaluation indicates that while the scoping of variables gives a very small improvement in performance of these two benchmarks (.05 % and .01 % respectively for *lbm* and *h264ref*), the different work distribution strategies do not change the performance at all.

But still the question remains whether there will be any significant performance change for other benchmarks due to the modification of the basic pragma. Previous work mentions that finding the best suited (OpenMP) pragma automatically in loops is non-trivial and needs programmer's support [45, 51, 24]. One approach for automatic modification of the pragmas for work sharing is to use machine-learning techniques [45, 51]. Auto-scoping of variables is still not supported in *bgxlc_r*.

Techniques used by the Oracle's Solaris compiler can be explored for auto-scoping [30].

5.6 Conclusion

In conclusion, this chapter shows that TLS is able to extract more parallelization from programs than traditional parallelization techniques such as OpenMP and SIMDization. Using single input dependence profiles, TLS candidate loops can be selected. However, there are different factors (*e.g.* number and coverage of loops, mispeculation overhead, L1 cache misses, dynamic instruction path length increase) that should be taken into consideration while selecting candidate loops for speculation. Benchmarks such as *lbm*, that have loops with good coverage and without dependence violation, are well suited for TLS. SPEC2006 and PolyBench/C benchmark poorly scale with increasing number of threads in BG/Q. This is expected because BG/Q was designed to run huge scientific applications that run for hours and even days, not for benchmarks with maximum running time of 30 minutes. While TLS can be applied to applications similar to the SPEC2006 and PolyBench/C benchmarks, the search remains open for an application that will have a much longer execution time and will have loops that are better suited for TLS execution in BG/Q than short-running benchmarks.

```
        double ux, uy, uz, u2, rho;

#pragma speculative for private( ux, uy, uz, u2, rho )

        SWEEP_START( 0, 0, 0, 0, 0, SIZE_Z )
                if( TEST_FLAG_SWEEP( srcGrid, OBSTACLE )) {
                        DST_C ( dstGrid ) = SRC_C ( srcGrid );
                        DST_S ( dstGrid ) = SRC_N ( srcGrid );
                        DST_N ( dstGrid ) = SRC_S ( srcGrid );
                        DST_W ( dstGrid ) = SRC_E ( srcGrid );
                        DST_E ( dstGrid ) = SRC_W ( srcGrid );
                        DST_B ( dstGrid ) = SRC_T ( srcGrid );
                        DST_T ( dstGrid ) = SRC_B ( srcGrid );
                        DST_SW( dstGrid ) = SRC_NE( srcGrid );
                        DST_SE( dstGrid ) = SRC_NW( srcGrid );
                        DST_NW( dstGrid ) = SRC_SE( srcGrid );
                        DST_NE( dstGrid ) = SRC_SW( srcGrid );
                        DST_SB( dstGrid ) = SRC_NT( srcGrid );
                        DST_ST( dstGrid ) = SRC_NB( srcGrid );
                        DST_NB( dstGrid ) = SRC_ST( srcGrid );
                        DST_NT( dstGrid ) = SRC_SB( srcGrid );
                        DST_WB( dstGrid ) = SRC_ET( srcGrid );
                        DST_WT( dstGrid ) = SRC_EB( srcGrid );
                        DST_EB( dstGrid ) = SRC_WT( srcGrid );
                        DST_ET( dstGrid ) = SRC_WB( srcGrid );
                        continue;
                }

                rho = + SRC_C ( srcGrid ) + SRC_N ( srcGrid )
                      + SRC_S ( srcGrid ) + SRC_E ( srcGrid )
                      + SRC_W ( srcGrid ) + SRC_T ( srcGrid )
                      + SRC_B ( srcGrid ) + SRC_NE( srcGrid )
                      + SRC_NW( srcGrid ) + SRC_SE( srcGrid )
                      + SRC_SW( srcGrid ) + SRC_NT( srcGrid )
                      + SRC_NB( srcGrid ) + SRC_ST( srcGrid )
                      + SRC_SB( srcGrid ) + SRC_ET( srcGrid )
                      + SRC_EB( srcGrid ) + SRC_WT( srcGrid )
                      + SRC_WB( srcGrid );

                ux = + SRC_E ( srcGrid ) - SRC_W ( srcGrid )
                     + SRC_NE( srcGrid ) - SRC_NW( srcGrid )
                     + SRC_SE( srcGrid ) - SRC_SW( srcGrid )
                     + SRC_ET( srcGrid ) + SRC_EB( srcGrid )
                     - SRC_WT( srcGrid ) - SRC_WB( srcGrid );
                uy = + SRC_N ( srcGrid ) - SRC_S ( srcGrid )
                     + SRC_NE( srcGrid ) + SRC_NW( srcGrid )
                     - SRC_SE( srcGrid ) - SRC_SW( srcGrid )
                     + SRC_NT( srcGrid ) + SRC_NB( srcGrid )
                     - SRC_ST( srcGrid ) - SRC_SB( srcGrid );
                uz = + SRC_T ( srcGrid ) - SRC_B ( srcGrid )
                     + SRC_NT( srcGrid ) - SRC_NB( srcGrid )
                     + SRC_ST( srcGrid ) - SRC_SB( srcGrid )
                     + SRC_ET( srcGrid ) - SRC_EB( srcGrid )
                     + SRC_WT( srcGrid ) - SRC_WB( srcGrid );

                ux /= rho;
                uy /= rho;
                uz /= rho;

                if( TEST_FLAG_SWEEP( srcGrid, ACCEL )) {
                        ux = 0.005;
                        uy = 0.002;
                        uz = 0.000;
                }

                u2 = 1.5 * (ux*ux + uy*uy + uz*uz);
                DST_C ( dstGrid ) = (1.0-OMEGA)*SRC_C ( srcGrid ) + DFL1*OMEGA*rho*(1.0                                  - u2);

                DST_N ( dstGrid ) = (1.0-OMEGA)*SRC_N ( srcGrid ) + DFL2*OMEGA*rho*(1.0 +       uy*(4.5*uy       + 3.0) - u2);
                DST_S ( dstGrid ) = (1.0-OMEGA)*SRC_S ( srcGrid ) + DFL2*OMEGA*rho*(1.0 +       uy*(4.5*uy       - 3.0) - u2);
                DST_E ( dstGrid ) = (1.0-OMEGA)*SRC_E ( srcGrid ) + DFL2*OMEGA*rho*(1.0 +       ux*(4.5*ux       + 3.0) - u2);
                DST_W ( dstGrid ) = (1.0-OMEGA)*SRC_W ( srcGrid ) + DFL2*OMEGA*rho*(1.0 +       ux*(4.5*ux       - 3.0) - u2);
                DST_T ( dstGrid ) = (1.0-OMEGA)*SRC_T ( srcGrid ) + DFL2*OMEGA*rho*(1.0 +       uz*(4.5*uz       + 3.0) - u2);
                DST_B ( dstGrid ) = (1.0-OMEGA)*SRC_B ( srcGrid ) + DFL2*OMEGA*rho*(1.0 +       uz*(4.5*uz       - 3.0) - u2);

                DST_NE( dstGrid ) = (1.0-OMEGA)*SRC_NE( srcGrid ) + DFL3*OMEGA*rho*(1.0 + (+ux+uy)*(4.5*(+ux+uy) + 3.0) - u2);
                DST_NW( dstGrid ) = (1.0-OMEGA)*SRC_NW( srcGrid ) + DFL3*OMEGA*rho*(1.0 + (-ux+uy)*(4.5*(-ux+uy) + 3.0) - u2);
                DST_SE( dstGrid ) = (1.0-OMEGA)*SRC_SE( srcGrid ) + DFL3*OMEGA*rho*(1.0 + (+ux-uy)*(4.5*(+ux-uy) + 3.0) - u2);
                DST_SW( dstGrid ) = (1.0-OMEGA)*SRC_SW( srcGrid ) + DFL3*OMEGA*rho*(1.0 + (-ux-uy)*(4.5*(-ux-uy) + 3.0) - u2);
                DST_NT( dstGrid ) = (1.0-OMEGA)*SRC_NT( srcGrid ) + DFL3*OMEGA*rho*(1.0 + (+uy+uz)*(4.5*(+uy+uz) + 3.0) - u2);
                DST_NB( dstGrid ) = (1.0-OMEGA)*SRC_NB( srcGrid ) + DFL3*OMEGA*rho*(1.0 + (+uy-uz)*(4.5*(+uy-uz) + 3.0) - u2);
                DST_ST( dstGrid ) = (1.0-OMEGA)*SRC_ST( srcGrid ) + DFL3*OMEGA*rho*(1.0 + (-uy+uz)*(4.5*(-uy+uz) + 3.0) - u2);
                DST_SB( dstGrid ) = (1.0-OMEGA)*SRC_SB( srcGrid ) + DFL3*OMEGA*rho*(1.0 + (-uy-uz)*(4.5*(-uy-uz) + 3.0) - u2);
                DST_ET( dstGrid ) = (1.0-OMEGA)*SRC_ET( srcGrid ) + DFL3*OMEGA*rho*(1.0 + (+ux+uz)*(4.5*(+ux+uz) + 3.0) - u2);
                DST_EB( dstGrid ) = (1.0-OMEGA)*SRC_EB( srcGrid ) + DFL3*OMEGA*rho*(1.0 + (+ux-uz)*(4.5*(+ux-uz) + 3.0) - u2);
                DST_WT( dstGrid ) = (1.0-OMEGA)*SRC_WT( srcGrid ) + DFL3*OMEGA*rho*(1.0 + (-ux+uz)*(4.5*(-ux+uz) + 3.0) - u2);
                DST_WB( dstGrid ) = (1.0-OMEGA)*SRC_WB( srcGrid ) + DFL3*OMEGA*rho*(1.0 + (-ux-uz)*(4.5*(-ux-uz) + 3.0) - u2);
        SWEEP_END
}
```

Figure 5.6: A loop from the SPEC2006 *lbm* benchmark that needs additional clauses added to the basic TLS pragma for better performance.

Chapter 6

Related Work

This chapter summarizes the related work on TLS and the use of profiling for speculation.

6.1 Thread Level Speculation

In one of the earlier works on TLS Franklin and Sohi propose Address Resolution Buffer (ARB), a hardware mechanism, ARB, to perform dynamic reordering of memory references [17]. ARB supports the following features: 1) dynamic disambiguation of memory references; 2) multiple memory references per cycle; 3) out-of-order execution of memory references; 4) unresolved loads and stores; 5) speculative loads and stores; and 6) memory renaming. ARB is a shared table that is used to track speculative loads and stores. The scheme allows speculative loads and speculative stores by keeping the uncommitted store values in the hardware structure and forwarding them to subsequent loads that require the value.

Following the work by Franklin and Sohi, multiple proposals have been made to move speculative data into each core's private cache or write buffer and to use cache-coherence protocols for memory disambiguation. In contrast to the ARB's centralized support for speculative versions, Vijaykumar *et al.* propose the *Speculative Versioning Cache (SVC)*, that uses distributed caches to eliminate the latency and bandwidth problems of ARB [47]. The SVC approach is based on unification of cache coherence and speculative versioning and uses an organization similar to snooping bus-based coherent caches. Memory references that get a hit in the private cache do not use the bus. The committed tasks do not write back the speculative versions of data all at one time. Instead, versions are marked, together, as 'committed' at commit time, without performing any data movement. Each cache line is individually handled when it is accessed the next time.

Hammond *et al.* describe the implementation of speculative execution in *Hydra*, a chip multiprocessor (CMP) [20]. Their approach is a hardware + software hybrid mechanism (introduction of a number of software-speculation control handlers and modifications to the shared secondary cache memory system of the CMP) to achieve TLS. They show that TLS is profitable for applications with substantial amount of medium-grained loops. When the granularity of parallelism is too small, or

the available *inherent* parallelism in the application is low, the overhead of the software speculation handlers overwhelms the potential performance benefits from TLS.

Steffan *et al.* propose and evaluate a TLS system that scales to a wide range of machine sizes [43]. Their strategy is an extension of writeback invalidation-based cache coherence [28]. The scheme has: 1) a notion of whether a cache line has been speculatively loaded and/or speculatively modified and 2) a guarantee that a speculative cache line will not be propagated to regular memory, and (3) that speculation will fail if a speculative cache line is replaced. They add three new speculative coherence messages and speculative cache states to the cache coherence protocol. They show that, using their model, applications scale from single-chip multiprocessors or simultaneously multi-threaded processors up to large-scale machines that might use single-chip multiprocessors.

Ceze *et al.* describe an approach to disambiguate memory references so that the dependences in the code can be checked during program execution and threads can be committed or rolled back accordingly [6]. They hash-encode a thread's access information in a *signature*, and then add operations that efficiently process sets of addresses from the signature. They employ a *Bloom-filter-based* compact representation of a thread's access information. Their mechanism is called *Bulk* operation because they operate on a set of addresses. They show that, despite its simplicity, *Bulk* has competitive performance with more complex schemes.

Steffan *et al.* show that the performance of a TLS system is dependent on the way threads communicate values among them [44]. They apply three different actions - value prediction, dynamic synchronization and hardware instruction prioritization to improve value communication among threads. Their technique first explores how *silent stores* (value of the memory location before the store is the same as the value of the location after the store) can be exploited within TLS. The TLS system avoids data-dependence violations and makes dependent store-load pair independent if the store is silent. In this way the TLS system reduces the coherence traffic as well as future update traffic. They also use some compiler heuristics to select loops for TLS (based on the execution time, number of iterations of the loop) and apply compiler optimizations (*e.g.* loop unrolling and reduction of critical forwarding path of values between loop iterations) that significantly improve TLS performance.

Loop Selection for Speculative Execution

Because loops are mostly targeted for speculative execution, different methodologies are proposed to select candidate loops for speculative execution.

Colohan empirically studies the impact of thread size on the performance of loops [11]. He employs different techniques to unroll loops to determine the best thread size per loop. He also proposes a run-time system to measure the run-time performance of loops and select each loop dynamically. Due to the run-time overhead, the system can only select loops without considering loop

nesting.

Olukotun *et al.* propose and evaluate a static loop selection algorithm that selects the best loops in each level of dynamic loop nest as possible candidates to be parallelized [36]. The algorithm computes the frequency with which each loop is selected as best loop and select the parallelized loops based on the computed frequencies. However, this technique is only used to guide the heuristic in context-insensitive loop selection. The performance estimation of loops is obtained directly from simulation, and does not consider the effect of different compiler optimizations on the loop's performance.

Chen and Olukotun propose a dynamic loop selection framework for Java program [7]. They use hardware to extract useful information, such as dependence timing and speculative state requirements; and then estimate the speedup for a loop. Their technique is similar to the run-time system proposed by Colohan and can only select loops within a simple loop nested program. Considering the global loop nesting relations and selecting the loops globally introduces significant overhead for the run-time system.

Marcuello *et al.* propose a thread spawning scheme that supports spawning threads from any point in the program [32]. They use profile information to identify appropriate thread spawning points with more emphasis on thread predictability.

Wang *et al.* propose different techniques to select loops for TLS [50]. They use a construct called *loop-graph* that describes the different nesting relations among loops. Only one loop from a loop nest is parallelized based on some heuristics. They use run-time profile information to measure the performance improvement of parallelizing a loop. They also apply graph-pruning to reduce the *selection problem size* (which is otherwise NP complete). Most importantly, they show that the dynamic behaviour of a loop is sensitive to its calling context (for some invocation, the loop may gain performance improvement while for some invocations, it may not). They attach this calling context information in the loop graph.

6.2 Profiling for Speculation

Ju *et al.* propose a unified framework to exploit both data- and control-speculation targeting specifically for memory latency hiding [23]. The speculation is exploited by hoisting *load* instructions across potentially aliasing *store* instructions or conditional branches. The framework also contains recovery model that relies on compiler-generated explicit recovery blocks. Their framework shows modest performance improvement in SPECInt95 benchmarks by shortening critical paths despite some side effects, such as recovery code penalty during mispeculation and higher register pressures. They use *edge* and *path* profiles to support control speculation.

Chen *et al.* propose a data-dependence profiler targeted at speculative optimizations [8]. They perform speculative Partial Redundancy Elimination (PRE) and code scheduling, using a naive profiler and speculative support provided through the ALAT (Advanced Load Address Table) in the

Itanium processors. Their non-sampling profiler has a slowdown of up to 100x, and they propose sampling techniques to overcome this problem. They use a shadow space with a simple hashing scheme to facilitate fast address comparison for detecting data-dependences.

Lin *et al.* propose a compiler framework that is based on a speculative SSA (Static Single Assignment) form to incorporate speculative information for both data and control speculation [29]. Speculative SSA integrates the alias information directly into the *intermediate representation* (IR) using explicit *may modify operator* and *may reference operator*. Their speculative analysis is assisted by both *alias profiles* and heuristic rules.

The *Multiscalar* compiler selects tasks by walking the Control Flow Graph (CFG) and accumulating basic blocks into tasks using a variety of heuristics [48]. The task selection methodology for the Multiscalar compiler is revisited by Johnson *et al.* [22]. Instead of using a heuristic to collect basic blocks into tasks, the CFG is annotated with weights and broken into tasks using a min-cut algorithm. These compilers assume special hardware for dispatching threads; they do not specify when a thread should be launched.

In SPSM, loop iterations are selected by the compiler as speculative threads [13]. SPSM uses a special instruction *fork* that allows the compiler to specify when tasks begin executing. In addition, SPSM recognizes the potential benefits from prefetching but proposes no techniques to exploit it.

Bhowmik and Franklin build a framework for speculative multithreading on the SUIF-MachSUIF platform [5]. Within this framework, they consider dependence-based task selection algorithms. Like Multiscalar, they focus on compiling the whole program for speculation, but allow the compiler to specify a spawn location as in SPSM.

Du *et al.* propose a cost-driven compilation framework to perform speculative parallelization [12]. The compiler uses dependence-profile for task selection and for partitioning speculative loops into serial and parallel portions. The profiler tracks both intra- and loop-carried- true dependences for speculative loops. Loop-carried-dependences are used to guide the partitioning of loop bodies into a serial and a parallel portion. Since dependences originated from the serial portion do not trigger roll-back in the parallel portion, the key part of their framework is to move source computation of frequent dependences (called violating candidate) to the serial portion through instruction reordering. A cost model is used to select the optimal loop partition, that is based on the size of serial portion and the mispeculation cost of the parallel portion. The mispeculation cost is computed by combining re-execution cost of individual nodes weighted by probabilities of carried-dependences (for violating candidates) and intra-iteration dependences (for others). The framework delivers a 8% speedup for the SPECInt2000 benchmarks.

Quinones *et al.* evaluate the *Mitosis* compiler for exploiting speculative thread-level parallelism [40]. Mitosis compiler uses both dependence- and edge-profiles for: 1) generating speculative precomputation slices (P-slice); and 2) selecting spawning pairs. P-slice predicts live-in values for speculative tasks and contributes to the serial portion of the speculative execution. To minimize

P-slice overhead while maximizing the accuracy, the compiler uses dependence- and edge-profiles to prune instructions in P-slices. To select spawning pairs, another profile analyzes the sequential execution trace to model the speculative execution time of each candidate spawning pair without considering inter-task memory conflicts. They show a 2.2x speedup for the OLDEN benchmarks on their simulator.

Wu *et al.* propose a cost model for compiler-driven task selection for TLS [52]. The model employs profile-based analysis of may-dependences to estimate the probability of successful speculation. Their profiling tool, *DProf*, is able to measure dependence probability and independence window of loops. They provide two techniques: (1) dependence windows (the number of iterations that can be executed in parallel without affecting the output) and (2) dependence clustering (regions of the loop's iteration space with large independence windows that makes that portion profitable to execute in parallel). One important finding in their work is that there is little variability in independence window width in the hot loops for the SPEC2000 benchmarks. Loops are either parallel or serial with an independence window width of 1.

Kim *et al.* propose a scalable approach to data-dependence profiling that addresses both run-time and memory overhead in a single framework [25]. Their technique, called *SD3*, reduces the run-time overhead by parallelizing the dependence profiling step itself. To reduce the memory overhead, they compress memory accesses that exhibit stride patterns and compute data-dependences directly in a compressed format. For stride detection, they use a *finite state machine* (FSM). SD3 reduces the run-time overhead by a factor of 9.7 on 32 cores and reduces memory consumption by 20x with a 16x speedup in profiling time. They also show that the dependence behaviour for different inputs *does not* change for OmpSCR benchmarks and changes very less (correlation 0.98 among the dependence pairs discovered in the loops) for SPEC2006 benchmarks.

POSH is a TLS compiler built on top of *gcc* [31]. POSH performs a partitioning of the code into TLS tasks and considers both subroutines and loops. POSH also uses a simple profiling pass to discard ineffective tasks. While choosing tasks beneficial for speculation, POSH considers the task size and the squash frequency that is obtained by simulation of the parallel execution. L2 cache misses are also considered by the profiler to perform prefetching analysis.

Vanka and Tuck use a set-based approach for data-dependence profiling [46]. Rather than tracking pair-wise dependences, they identify important dependence relationships at compile time, group the relationships into sets, and then construct and operate directly on the sets at run-time. They use *Signatures*, an efficient set representation based on Bloom filters, to to trade-off performance and accuracy.

Though previous work [25, 14, 45] mention that there is little change in the variability of dependence behaviour of loops with respect to different inputs, there has not been an extensive study to support the claim. Therefore, there is no prior work to provide a solution to combine data-dependence profiles for loops that have varied dependence behaviour with different inputs (for ex-

ample, the loop in the Convex Hull algorithm as described in Section 3.7 of Chapter 3). Also there is no previous work on a detailed performance evaluation of TLS on real hardware and the different factors that affect TLS. From the above two prospects, this research makes new contributions.

Chapter 7

Conclusion

Thread Level Speculation (TLS) is a promising technique to extract parallelism from a program by providing the guarantee of correct execution even in the presence of dependences. Data-dependence profiling is necessary for TLS because the mispeculation overhead may degrade the performance of a program. But profile obtained from a single training run of the program is often not sufficient because Berube *et al.* [4] have shown that a program's behaviour may change depending on the input. Therefore the question remained open whether the dependence behaviour of loops changes based on the input to the program.

To find an answer to the question, this research presented an implementation of a combined dependence-profiling framework in the LLVM compiler infrastructure. A decision was made to keep as much dependence information as is necessary to perform a profitability analysis to find loops that are speculation candidate.

Previous work [25, 45] found some evidence that the dependence behaviour of loops does not change based on inputs. In this research, a wide range of benchmarks was studied to investigate that the previous claim that for most applications, a loop's dependence behaviour *does not* change with respect to inputs. This additional evidence will allow the future research in TLS to consider single input data-dependence profiles to predict the dependence behaviour of a loop.

Despite the finding that a single-input profile is sufficient to predict the dependence behaviour of a loop for most real-world applications, our investigation discovered a particular application (incremental algorithm to build Convex Hull of a two-dimensional set of points [10]) that has a loop with varied dependence behaviour across inputs. Executing the loop in the Convex Hull application speculatively with different inputs in IBM's BG/Q machine showed how the varied percentages of dependences occurring during runtime have an impact on speculative execution. As more applications with similar behaviour are discovered, the proposed combined dependence-profiling methodology will become useful to perform a cost analysis to find speculation candidate loops based on their probability of dependence.

Based on the finding that the loop's dependence behaviour does not change based on input, a performance evaluation of TLS applied to the SPEC2006 and PolyBench/C benchmarks was pre-

sented. Some simple heuristics that use single-input data-dependence profiles, were applied for the performance evaluation. This research is the first performance evaluation of the TLS implementation in the BG/Q.

The automatic speculative parallelization framework, SpecEval, described in Chapter 4, performs a single-input data-dependence profiling and uses the profile file to parallelize loops through source-code annotation (TLS-specific pragma insertion). To find the *may dependences* inside the loops of these benchmarks, two different static dependence analyzers were used: (1) *Polly* — the polyhedral dependence analyzer [19]; and (2) the loop dependence analyzer in LLVM.

The experiments with Polly indicated that the coverage of the loops and the materialization of *may-dependences* during run-time have impact on the TLS performance of the SPEC2006 and PolyBench/C benchmarks. To study the impact of mispeculation, loops were speculatively executed in parallel regardless of their coverage. To study the impact of profiling, loops were executed speculatively in parallel even if the *may dependences* inside them materialize during run-time. The main finding was that thread creation overhead and the mispeculation overhead slows down the TLS performance of some benchmarks.

However, filtering *cold* loops and loops with run-time dependence gave some performance improvement when applied with the automatic OpenMP parallelizer of Polly. Using SpecEval, 4x and 8.1x speedup were achieved for *lbm* and *gemm* respectively for 8 threads over the sequential version compiled at the lowest optimization level (-O0) of the *bgxlc_r* compiler. Results also showed that only a few SPEC2006 benchmarks (*lbm*, *milc*, *povray* and *bzip2*) scaled with increasing number of threads, indicating that there is limited speculative parallelism available for SPEC2006 benchmarks.

In the experiments with LLVM's built-in dependence analyzer, it was evaluated how the SPEC2006 and PolyBench/C benchmarks perform when TLS is applied along with the existing auto-parallelizers of the *bgxlc_r* compiler, with the sequential code optimized at the highest optimization level (-O5) as baseline . A study of performances of three parallel versions of the code such as: (1) automatically SIMDized and (2) OpenMP + SIMD parallelized code by the bgxlc_r compiler and (3) automatic SIMDized + OpenMP parallelized + speculatively parallelized code; was done.

As expected, the speedups achieved from the three parallel versions were significantly lower when the baseline was sequential code optimized at the highest optimization level, indicating the high impact of sophisticated optimization techniques (*e.g.* loop unrolling, constant propagation, inter-procedural optimizations etc.) that are turned on at the -O5 level. However, some benchmarks (*h264ref*, *sjeng*, *jacobi*, *cholesky* and *dynprog*) experienced a slowdown due to TLS. The different factors found to impact the TLS performance are as follows.

First, the number of loops and their coverage were found to have impact on TLS performance. For instance, *lbm*'s performance improves under TLS due to the high coverage of speculative loops but the TLS performance of *sjeng*, *bzip2*, *mcf* were negatively affected by the speculative execution of cold loops.

Second, the mispeculation overhead due to speculative execution of loops with function calls that introduce dependences during run-time had an impact on TLS performance. The newly introduced dependences are not easily detected with off-line profiling due to the complexity of storing the calling context for each *callee* and performing an inter-loop dependence analysis when the *callee* has loops within.

Third, the flushing of the L1 cache before entering a speculative region for the *Long-running* (LR) mode in BG/Q had an impact on the execution of benchmarks (*cholesky* and *dynprog*). There is a need of support for *Short-running* (SR) TLS in BG/Q because it is necessary for these types of applications. The SR implementation is the future plan of this research.

Lastly, the dynamic-path-length increased significantly due to speculative execution of loops with a small iteration count that construct most portion of the code. These loops are not good candidates for TLS because the thread creation overhead limits the performance of these benchmarks. In future TLS research, these factors should be considered while selecting TLS candidate loops.
Currently we are working on the implementation of TLS in the LLVM compiler. LLVM has already been ported to BG/Q by Hal Finkel from Argonne National Laboratory. The plan is to implement short-running (SR) mode along with the existing long-running (LR) mode for TLS and then use heuristics that consider the aforesaid factors that impact TLS performance to select speculation candidate loops.

Bibliography

[1] SPEC2006 Benchmark suite. http://www.spec.org/cpu2006/.

[2] A.J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, 1966.

[3] Paul Berube. *Methodologies for May-Input Feedback-Directed Optimization*. PhD thesis, University of Alberta, Fall 2012.

[4] Paul Berube, Adam Preuss, and José Nelson Amaral. Combined profiling: practical collection of feedback information for code optimization. In *Proceedings of the second joint WOSP/SIPEW International Conference on Performance engineering*, pages 493–498, Karlshure, Germany, 2011.

[5] Anasua Bhowmik and Manoj Franklin. A general compiler framework for speculative multithreading. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures(SPAA)*, pages 99–108, Winnipeg, Manitoba, Canada, 2002.

[6] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, pages 227–238, Boston, MA, USA, 2006.

[7] Michael K. Chen and Kunle Olukotun. The JRPM system for dynamically parallelizing Java programs. In *International Symposium on Computer Architecture (ISCA)*, pages 434–446, San Diego, California, 2003.

[8] Tong Chen, Jin Lin, Xiaoru Dai, Wei-Chung Hsu, and Pen-Chung Yew. Data dependence profiling for speculative optimizations. In *Compiler Construction (CC)*, pages 57–72, Barcelona, Spain, Mar-Apr 2004.

[9] Marcelo Cintra, Jose Martinez, and Josep Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, pages 13–24, Jun 2000.

[10] Kenneth L. Clarkson, Kurt Mehlhorn, and Raimund Seidel. Four results on randomized incremental constructions. *Comput. Geom. Theory Appl.*, 3(4):185–212, September 1993.

[11] Christopher Colohan. The Impact of Thread Size and Selection on the Performance of Thread-Level Speculation. In *Systems Design and Implementation seminar*, CMU, Feb 2003.

[12] Zhao-Hui Du, Chu-Cheow Lim, Xiao-Feng Li, Chen Yang, Qingyu Zhao, and Tin-Fook Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Programming Language Design and Implementation (PLDI)*, pages 71–81, Washington DC, USA, 2004.

[13] Pradeep K. Dubey, Kevin O'Brien, Kathryn M. O'Brien, and Charles Barton. Single-program speculative multithreading (SPSM) architecture: compiler-assisted fine-grained multithreading. In *International Conference on Parallel Architectures and Compilation Techniques(PACT)*, pages 109–121, Limassol, Cyprus, 1995.

[14] K.-F. Faxen, K. Popov, L. Albertsson, and S. Janson. Embla - Data Dependence Profiling for Parallel Programming," Complex, Intelligent and Software Intensive Systems. In *International Conference on Complex, Intelligent and Software Intensive Systems*, pages 780–785, Mar 2008.

[15] P. Feautrier. Array expansion. In *International Conference on Supercomputing (ICS)*, July 1988.

[16] P Feautrier. Some efficient solutions to the affine scheduling problem. part 2. multidimensional time. *International Journal of Parallel Programming (IJPP)*, 21(6):389–420, 1992.

[17] Manoj Franklin and Gurindar S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transaction on Computers*, 45(5):552–571, May 1996.

[18] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming (IJPP)*, 34(3):261–317, July 1988.

[19] Tobias Grosser, Hongbin Zheng, Raghesh A, Andreas Simbürger, Armin Grosslinger, and Louis-Noël Pouchet. Polly - Polyhedral optimization in LLVM. In *International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Chamonix, France, Apr 2011.

[20] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *International Conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 58–69, San Jose, California, USA, 1998.

[21] R.A. Haring, M. Ohmacht, T.W. Fox, M.K. Gschwind, D.L. Satterfield, K. Sugavanam, P.W. Coteus, P. Heidelberger, M.A. Blumrich, R.W. Wisniewski, A. Gara, G.L.-T. Chiu, P.A. Boyle, N.H. Chist, and Changhoan Kim. The IBM Blue Gene/Q compute chip. *Micro, IEEE*, 32(2):48 –60, March-April 2012.

[22] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Programming Language Design and Implementation (PLDI)*, pages 59–70, Washington DC, USA, 2004.

[23] Roy Dz-ching Ju, Kevin Nomura, Uma Mahadevan, and Le-Chun Wu. A Unified Compiler Framework for Control and Data Speculation. In *International Conference on Parallel Architectures and Compilation Techniques(PACT)*, pages 157–168, Philadelphia, Pennsylvania, USA, 2000.

[24] M. Kim, H. Kim, and C. Luk. Prospector: A dynamic data-dependence profiler to help parallel programming. In *Proceedings of the USENIX workshop on Hot Topics in parallelism (HotPar)*, 2010.

[25] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. SD3: A Scalable Approach to Dynamic Data-Dependence Profiling. In *International Symposium on Microarchitecture(MICRO)*, pages 535–546, 2010.

[26] Xiangyun Kong, David Klappholz, and Kleanthis Psarris. The I Test: A New Test for Subscript Data Dependence. In *International Conference on Parallel Processing (ICPP)*, pages 204–211, 1990.

[27] Chris Lattner. The LLVM Compiler Infrastructure. http://llvm.org.

[28] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. *ACM*, 18(3a), 1990.

[29] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. A compiler framework for speculative analysis and optimizations. In *Programming Language Design and Implementation (PLDI)*, pages 289–299, San Diego, California, USA, Jun 2003.

[30] Y. Lin, C. Terboven, D. Mey, and N. Copty. Automatic scoping of variables in parallel regions of an openmp program. *WOMPAT*, 3349:83–97, 2004.

[31] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: a TLS compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming(PPoPP)*, pages 158–167, New York, New York, USA, 2006.

[32] Pedro Marcuello, Antonio González, and Jordi Tubella. Speculative multithreaded processors. In *International Conference on Supercomputing (ICS)*, pages 77–84, Melbourne, Australia, 1998.

[33] M.Wolfe. High performance compilers for parallel computing. Addison Wesley, 1996.

[34] NASA. NAS parallel benchmarks. http://www.nas.nasa.gov/publications/npb.html.

[35] University of Maryland and Intel. Biobench/bioparallel: A benchmark suite for bioinformatics applications. http://www.ece.umd.edu/biobench/.

[36] K. Olukotun, L Hammond, and M Willey. Improving the Performance of Speculatively Parallel Applications on the Hydra CMP. In *International Conference on Supercomputing (ICS)*, pages 21–30, Rhodes, Greece, June 1999.

[37] Venkatesan Packirisamy, Antonia Zhai, Wei-Chung Hsu, Pen-Chung Yew, and Tin-Fook Ngai. Exploring speculative parallelism in SPEC2006. In *International Symposium on Performance Analysis of Systems and Software(ISPASS)*, pages 77–88, 2009.

[38] Louis-Noël Pouchet. PolyBench/C Benchmarks. http://www.cse.ohio-state.edu/ pouchet/software/polybench/.

[39] W Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *International Conference on Supercomputing (ICS)*, pages 4–13, Aug 1991.

[40] Carlos Garcia Quinones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Programming Language Design and Implementation (PLDI)*, pages 269–279, Chicago, IL, USA, 2005.

[41] Lawrence Rauchwerger and David Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Programming Language Design and Implementation (PLDI)*, pages 218–232, 1995.

[42] B. Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming languages (POPL)*, pages 21–24, Jan 1996.

[43] J. Greggory Steffan, Christopher Colohan, Antonia Zhai, and Mowry Todd. A scalable approach to thread-level speculation. In *International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2000.

[44] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. Improving Value Communication for Thread-Level Speculation. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 65–76, Cambridge, Mass,USA, 2002.

[45] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *Programming Language Design and Implementation (PLDI)*, pages 177–187, Dublin, Ireland, 2009.

[46] Rajeshwar Vanka and James Tuck. Efficient and accurate data dependence profiling using software signatures. In *Code Generation and Optimization (CGO)*, pages 186–195, San Jose, California, 2012.

[47] Gopal Vijaykumar, Smith, and Sohi. Speculative versioning cache. In *International Symposium on High-Performance Computer Architecture*, pages 195–205, Feb 1998.

[48] T. N. Vijaykumar and Gurindar S. Sohi. Task selection for a multiscalar processor. In *International Symposium on Microarchitecture(MICRO)*, pages 81–92, Dallas, Texas, USA, 1998.

[49] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of blue Gene/Q hardware support for transactional memories. In *PACT*, pages 127–136, 2012.

[50] Shengyue Wang, Xiaoru Dai, KiranS. Yellajyosula, Antonia Zhai, and Pen-Chung Yew. Loop Selection for Thread-Level Speculation. *Languages and Compilers for Parallel Computing*, 4339:289–303, 2006.

[51] Zheng Wang and Michael F.P. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *Principles and practice of parallel programming(PPoPP)*, pages 75–84, 2009.

[52] Peng Wu, Arun Kejariwal, and Călin Caşcaval. Compiler-Driven Dependence Profiling to Guide Program Parallelization. *Languages and Compilers for Parallel Computing*, pages 232–248, 2008.

[53] Polychronis Xekalakis, Nikolas Ioannou, and Marcelo Cintra. Combining thread level speculation helper threads and runahead execution. In *International Conference on Supercomputing (ICS)*, pages 410–420, 2009.

Appendix A

TLS Specific Compiler Pragmas in bgxlc_r

The following clauses can be used along with the '#pragma speculative for' of the bgxlc_r compiler[1].

A.1 #pragma speculative for

The *speculative for* directive instructs the compiler to speculatively parallelize a for loop.

A.1.1 Syntax

The pragma can be used as *#pragma speculative for clause* where *clause* can be one of the following:

- **default (shared/none):** Defines the default data scope of variables in each thread. Specifying **default (shared)** is equivalent to stating each variable in a **shared (list)** clause. Specifying **default (none)** requires that each data variable visible to the parallelized statement block must be explicitly listed in a data scope clause, with the following exceptions:

  1. The variables are const-qualified.
  2. The variables are specified in an enclosed data scope attribute clause.
  3. The variables are used as a loop control variable referenced only by a corresponding *speculative for* directive.

- **shared (list):** Declares the scope of the comma-separated data variables in list to be shared across all threads.

- **private (list):** Declares the scope of the data variables in list to be private to each thread. Data variables in list are separated by commas.

- **firstprivate (list):** Declares the scope of the data variables in list to be private to each thread. Each new private object is initialized with the value of the original variable as if there was

[1] Available at http://www-01.ibm.com/support/docview.wss?uid=swg27027065&aid=1

an implied declaration within the statement block. Data variables in *list* are separated by commas.

- **lastprivate (list):** Declares the scope of the data variables in list to be private to each thread. The final value of each variable in *list*, if assigned, is the value assigned to that variable in the last iteration. Variables not assigned a value have an indeterminate value. Data variables in *list* are separated by commas.

- **num_threads (int_exp):** The value of *int_exp* is an integer expression that specifies the number of threads to use for the parallel region. If dynamic adjustment of the number of threads is also enabled, then *int_exp* specifies the maximum number of threads to be used.

- **reduction (operator: list):** Performs a reduction on all scalar variables in list using the specified operator. Reduction variables in list are separated by commas. *operator* can be one of: +, -, |, ^, ||, *, & and &&.

- **schedule (type):** Specifies how iterations of the for loop are divided among available threads. Thread-level speculative execution supports static scheduling. Acceptable values for type are as follows:

  1. **static:** Iterations of a loop are divided into chunks of size:
     ceiling(number_of_iterations/number_of_threads).
     Each thread is assigned a separate chunk. This scheduling policy is also known as *block scheduling*.

  2. **static,n:** Iterations of a loop are divided into chunks of size *n*. Each chunk is assigned to a thread in *round-robin* fashion. *n* must be an integral assignment expression of value 1 or greater. This scheduling policy is also known as *block cyclic scheduling*. If n=1, iterations of a loop are divided into chunks of size 1 and each chunk is assigned to a thread in round-robin fashion.

A.1.2 Usage

The directives for thread-level speculative execution only take effect if the user specifies the *-qsmp=speculative* compiler option. The speculative for directive must immediately precede a for loop. The user cannot create a local object of variable length array within a thread-level speculative execution region.