Evaluation of Thread Level Speculation in BlueGene/Q

Arnamoy Bhattacharyya
Department of Computing Science
University of Alberta
Edmonton, Alberta

José Nelson Amaral
Department of Computing Science
University of Alberta
Edmonton, Alberta

Hal Finkel
Argonne National Laboratory
Argonne, Illinois

July 29, 2014

Abstract

Thread Level Speculation(TLS) is a hardware/software technique that guarantees correct parallel execution of loops even in the presence of dependence and has potential to lead to performance gains through the parallelization of loops that cannot be proven to be free of dependencies at compile time. However, given the overhead of TLS execution, the selection of loops to be speculated is important to avoid performance degradation. Data-dependence profiling is often used to find out if the may dependencies reported by the static analysis of a compiler materialize at runtime. A cost analysis may conclude that some loops with a lower probability of dependence should be speculatively parallelized. This report addressed the question as to whether a loops' dependence behaviour changes when the input to the program changes — a study of 57 different benchmarks indicates that it usually does not change. Then the report describes SpecEval, a new automatic speculative parallelization framework that uses single-input data-dependence profiles to find speculation candidates in the SPEC2006 and PolyBench/C benchmarks. This report also presents the first performance evaluation of TLS implementation in IBM's BlueGene/Q supercomputer and shows that the performance of TLS is affected by several factors, including: number and coverage of speculated loops, miss-speculation overhead due to function calls in a speculated loop, L1 cache miss rate and dynamic instruction path length affects.

0.1 Introduction

Existing auto-parallelizers must follow a conservative approach and generate sequential code for loops with potential dependencies. 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 loops. A parallelizing 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-analyses* check whether the same address may be referenced (loaded from or stored into) by different iterations of a loop. If two different iterations of the loop access the same memory address, the loop contains a loop-carried dependence and it is not parallelized.

Many existing static dependence analyses can be used by a parallelizing compiler to determine which loops are candidates for parallelization. If the compiler cannot determine, at compile time, whether there will be a dependence at run-time, *data-dependence profiling* can be used to predict the actual occurrence of dependencies at run-time [1, 2]. Data-dependence profiling records the memory addresses accessed by possibly dependent load/store instructions for all *may*-dependencies. This run-time dependencies for a *training* run of the program is saved to a profile file and the number of actual dependencies recorded — hopefully for multiple runs with different data inputs — is used to predict the materialization of such dependencies in future runs of the same program. A parallelization that relies on data-dependence profiling must execute the parallel code speculatively. The speculative execution provide a safe fall-back execution path for the runs in which profile-based predictions of absence of dependence turn out to be incorrect. Until recently, there were limited options for this fall-back path. Examples include the fall-back code for advanced loads in the Intel IA64 architecture and software solutions based on a combination of complier-generated code and runtime functionality. Recently, however, hardware-supported thread-level speculation became a reality, thus offering the potential for an easier-to-implement and more efficient fall-back path [3].

An interesting research question is whether this new fall-back path may lead to more opportunities for parallelization. For example, when a static dependence analysis, relying on imprecise alias information, reports that a load-store pair *may* be dependent at run time the loop still can be executed in parallel if either the dependence does not actually materialize during run-time, or if there is a guarantee that the parallel execution of the loop will not affect the correctness of the program when the dependence does occur at run time.

Thus, two important question are in front of us. (1) How effectively can the TLS support be used to improve the performance of standard benchmarks? (2) Do data-dependence profiling predictions vary with the set of inputs used for the profiling runs? Past work has suggested that, for widely used sets of benchmarks, the dependence behaviour does not change with the data input [4, 5, 6]. This result is confirmed by our own study described in this report and it is great news because it indicates that, for a wide class of applications, an accurate data-dependence prediction can be made based on a single profiling run that executes the loop of interest. However, past proposals to use this invariability did not provide a safety net to ensure that all possible runs of the program would produce the correct result. One of the main contributions of this report is to combine this simpler data-dependence profiling prediction with TLS to provide a framework for speculative parallelization.

Using the TLS-enabled IBM's BlueGene/Q supercomputer as an experimental platform, this research makes the following contribution:

- A description of *SpecEval*, a new automatic speculative parallelization framework that combines LLVM with the IBM bgxlc_r compiler to evaluate the performance of TLS in the BG/Q. *SpecEval* uses single-input data-dependence profile to find loops that are candidates for speculation in the SPEC2006 and PolyBench/C benchmarks.

- A detailed study using 57 benchmarks that confirms the previous claim that loop-dependence behaviour does not change based on program input.

- The first study of the performance impact of TLS when applied along with the existing auto-parallelizers of the *bgxlc_r* compiler (SIMDizer and OpenMP parallelizer).

- A detailed study of different factors that impact the speculative execution of loops in the BG/Q.

0.2 Related Work

Architecture techniques to support TLS have been extensively studied and found to be successful due to their lower overhead. Multiscalar [7] introduced the concept of hardware-based TLS and initially used hardware buffers called Address Resolution Buffers (ARB) [8]. Later studies relied on shared-memory cache coherence protocols to support TLS [9, 10, 11, 12, 13, 14]. There has been proposals on software-only TLS [15, 16] and the software solution revealed very high overhead [17, 18].

Several profiling and speculation frameworks have been proposed [2, 19, 20, 1, 21, 22]. *Embla* is a data-dependence profiler that supports TLS [6]. Chen *et al.* propose a data-dependence profiler developed for speculative optimizations [1]. They perform speculative Partial Redundancy Elimination (PRE) and code scheduling using a naive profiler and the speculative support provided through the Advanced Load Address Table (ALAT) in the Itanium processors. Wu *et al.* use the concept of independence windows and dependence clustering to find opportunities for speculative parallelization [23]. The proposed profiler, called *DProf*, performs iteration-grain disambiguation and differentiates between intra- and inter-iteration dependencies. POSH is a TLS compiler that uses a simple profiling pass to discard ineffective tasks. The criteria to select a task for TLS in POSH includes the task size, the expected squash frequency — obtained by simulation of the parallel execution — and L2 cache misses [24]. There has been also a limit study of TLS [25]. There has been work towards shadow memory-based approaches to speculation [26].

Previous research found that there is limited variability in the dependence behaviour of loops across inputs for a few benchmarks [4, 5, 6]. In this study, an evaluation of a wide range of benchmarks, which have been used to evaluate the potential of TLS confirms those findings. Earlier research proposals for hardware/hardware+software TLS used simulation to predict the performance of such systems. To the best of our knowledge this is the first evaluation of the actual performance of a hardware-supported TLS system.

0.3 Input Sensitivity in Data dependence Profiling

This section examines the sensitivity, to variations in the program data input, of the data-dependence prediction obtained from profiling in loops from 57 different benchmarks from the SPEC2006, PolyBench/C, Biobenchmarks and NAS benchmark suites with different inputs as listed in Table 1. Confirming the observation of previous studies [4, 5, 6], none of these benchmarks have a loop where variations on dependence behaviour across inputs was observed. For loops that were signalled as containing may-dependencies by the compiler, either the dependence materializes for all program inputs or it materializes for none of them. A possible inference from this result is that the may-dependence is due to the imprecision of the static dependence analysis rather than actual variations in the dependence behaviour of the program. This finding is promising for TLS research because it indicates that costly many-input profiling is seldom required. The next section describes an automatic speculative-parallelization framework that uses single-input data-dependence profiling to find loops that are candidates for speculation.

0.3.1 When Data Input Affect Dependences: A Case Study

Even though applications with input-determined dependence behaviour are rare, they do existent. This section presents a case study of such an application — the construction of a 2D-Hull — and explores the impact of different dependence densities, resulting from different inputs, on the speedup of such applications when TLS is used. Automatic TLS parallelizers should use a cost model that takes into account the probability distribution of dependences in the loops such applications. This case study illustrate a case where the application data input determines this probability.

Table 1: Benchmarks used in the input dependence behaviour study. BB = Biobenchmark

Suite	Benchmark	Suite	benchmark	Suite	Benchmark	Suite	Benchmark
SPEC2006	lbm	PolyBench/C	2mm	NAS	BT	BB	mummer
	h264ref		3mm		CG		protpar
	hmmer		gemm		DC		dnamove
	mcf		gramschmidt		EP		dnacomp
	sjeng		jacobi		FT		dnaml(2)
	sphinx3		lu		IS		dnamlk
	bzip2		seidel		LU		dnadist
	milc		cholesky		MG		dolmove
	gobmk		dynprog		SP		restml(2)
	namd		fdtd_2d		UA		kitsch
BB	gendist	BB	tigr	BB	clustalw	BB	dynprog
	hmmer		protdist		dnapars		dnapenny
	dnainvar		seqboot		dnamlk2		dollop
	dolpenny		fitch		neighbor		

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 and due to Clarkson *et al.* [27], computes the convex hull, *i.e.* the smallest enclosing polygon, of a set of points in a plan. The input to Clarkson's algorithm is a set of (x, y) point coordinates. The algorithm starts with a triangle composed by an initial set of three points and adds points. If the new point lies inside the current solution, it is discarded. Otherwise, a new convex hull is computed. Any change to the solution found so far causes a dependence that was classified as a may dependence to become a true dependence 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 has a dependence with probability in $\phi\left(\frac{\sqrt{i}}{i}\right)$. If the points lie uniformly on a square, the probability of a dependence is in $\phi\left(\frac{\log i}{i}\right)$.

This case studied used four different input sets. Each input contains 10-million points.

- **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.
- **Square:** has an uniform distribution of points inside a square.
- **Disc:** is an uniform distributions of points inside a disc.
- **Circle:** distributes all the points around a circle.

The Kuzmin input set leads to very few dependence violations because points far from the center are very scarce. The Square input set leads to an enclosing polygon with fewer edges than the Disc input set, thus generating fewer dependence violations. The Circle input set leads to a very large number of dependence violations.

The different inputs affect the dependence behaviour of the main loop of the Convex hull application. The percentage of dependence violations is 0.001% for Kuzmin, 0.005% for Square, 0.035% for Disc, and 10.4% for Circle. Figure 1 shows the effect of the dependence materialization at runtime on the percentage change in execution time of the Oracle version of TLS with respect to the baseline described in Section 0.5. In this version of TLS the loops in the 2d-hull were executed speculatively in parallel irrespective of the different probabilities of dependences materialization. These loops are not automatically parallelized by the SIMD or OpenMP parallelizer because the compiler can not prove the absence of dependence at compile time.

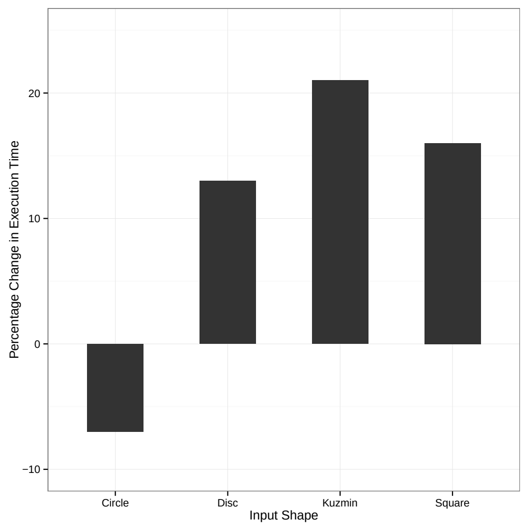

Figure 1: Percentage change in execution time for applying TLS for the 2d-hull application

This case study illustrates that in some applications, a TLS framework that takes into consideration the correlation between the input dataset and the variability of dependence materialization will have an effect in performance. The input-sensitive data profiling strategy described in this Section will be useful in such cases.

0.4 SpecEval: An Automatic Speculative Parallelization Framework

This section presents *SpecEval*, an evaluation framework for speculative parallelization that uses LLVM passes to find *may dependencies* inside loops, to profile and to generate debug information so that source-code instrumentation can be performed. *SpecEval* then uses bgxlc_r to produce executable code from the instrumented source code. Even though our group has access to the source code of the XL compiler for other research initiatives, for this research such access was not possible because the goal is to produce a TLS-enabled path in LLVM, an open-source compiler. Thus *SpecEval* is only an evaluation framework, rather than a compilation solution, to assess the potential for the use of TLS in the BG/Q. A shown in Figure 2, *SpecEval* has three phases:

1. **Collection of profile and debug information:** The first step is the compilation of the source code to Intermediate Representation (IR) with the `-g` option of the LLVM compiler so that debug information that maps executable code to source code can be collected. This debug information allows *SpecEval* to determine the source-code file name and line number where each speculative loop appears so that an speculation pragma can be inserted. Step 2 runs a dependence-analysis pass to find *may dependencies*. In step 3, a newly written instrumentation pass inserts calls to functions of a newly written library to prepare the code for profiling. Step 4 uses a newly written LLVM pass to collect the source-code file name and line number for all loops in the program. This debug information is stored in a file indicated as *Loop log* in Figure 2. Step 5 executes the instrumented IR produced at step 3 with a training input to collect the data-dependence profile.

2. **Source-code instrumentation:** A newly written C program takes the *Loop log* and the data-dependence profile file as inputs and inserts speculative pragmas in the source code before *for* loops that are found to be speculation candidates based on a heuristic. The heuristic selects a loop for

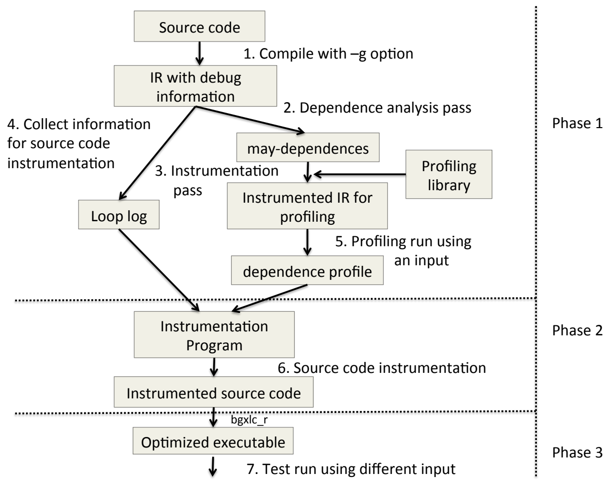
Source code
1. Compile with –g option
IR with debug information
2. Dependence analysis pass
4. Collect information for source code instrumentation
may-dependences
Profiling library
3. Instrumentation pass
Instrumented IR for profiling
Loop log
5. Profiling run using an input
dependence profile
Phase 1
Instrumentation Program
6. Source code instrumentation
Instrumented source code
Phase 2
bgxlc_r
Optimized executable
7. Test run using different input
Phase 3

Figure 2: The *SpecEval* Framework.

speculation if the may dependencies of the loop are predicted to not materialize, through profiling, and the loop takes more than 1.2% of the whole program execution time. This threshold was selected empirically to prevent slowdowns due to the overhead of speculating loops that represent an insignificant portion of the program execution time. The TLS pragma — '*#pragma speculative for*', is used signal to the XL compiler that the loop should be speculated. This BG/Q-specific pragma divides the total iteration space into chunks of:

$$number\_of\_iterations/number\_of\_threads$$

Listing 1 shows a sample instrumented loop. For a given loop-nest of n-dimensions, if the pragma is applied to multiple loop levels, bgxlc_r automatically flattens the pragma to be effective on the outermost loop.

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 on the BG/Q.[1] A different input is used in the *profiling* run than the input used for the *test* run.

Listing 1: A *for* loop with the annotated speculative pragma in BlueGene/Q

```
#pragma speculative for
for (i =0; i < n; i++)
        //do something
```

This framework leveraged the dependence analysis and the versatility of the LLVM framework [28] and the compiler and runtime infrastructure developed by IBM to enable a first performance evaluation of the hardware support for TLS on the BG/Q.

0.5 Experimental Results

This section evaluates the impact of applying profile-driven TLS to the SPEC2006 and PolyBench/C benchmarks using *SpecEval*. The above two benchmarks are chosen because they are widely used before in the

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

Table 2: Hardware Details of Vesta

Feature	Details	Feature	Details
Racks	2 Rack, 1024 Node per rack	L2 Cache	Centrally shared, 32 MB
Node	16 PowerPC A2 Cores with 16GB RAM per node	L1 Cache	16KB I/D
I/O	32 I/O Nodes Per Rack	SMT	4-way SMT
Peak Performance	419.44 TF	Speed	1.6 GHz
Network Topology	5D Torus	Architecture	64 Bit

TLS literature [5, 29, 23]. Times reported are an average execution time from 60 runs. For all experiments, a one sample Student's t-test performed on the 60 execution times shows that the p-values are in the range between 0.15-0.33 — a p-value less than or equal to 0.05 would indicate a significant variation.

Previous studies have used TLS for benchmarks with many parallel loops without regards for the nature of potential dependencies [30, 17]. However, there is no point on applying TLS to a loop that is known to not have dependencies — such loops should be executed in parallel rather than speculatively — or to a loop that is known to have an actual loop-carried dependence — the speculation of such loops is certain to fail. Therefore *SpecEval* only applies TLS to loops for which the compiler reported *may dependence* and where no true loop-carried dependencies exist. Therefore the opportunities for TLS under such assumptions are diminished by the quality of the dependence analysis in the compiler. *SpecEval* currently rely on the loop dependence analysis in LLVM. A similar approach can be used for other compilation frameworks in the future.

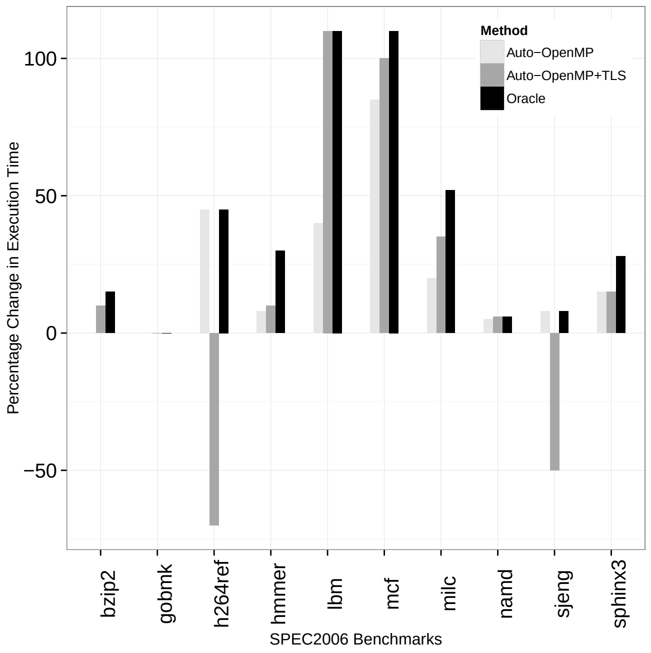

Figure 3: Percentage change in execution time from different parallelization techniques for SPEC2006 benchmarks over the auto-SIMDized code. The OpenMP and TLS versions use 4 threads.

0.5.1 Effect of applying TLS with AutoSIMD and AutoOpenMP parallelizer on the bgxlc_r

The baseline for comparison is an executable generated by the bgxlc compiler from IBM at the highest level of optimization (-O5) with auto-SIMDization. This code is generated with the following command: `bgxlc -O5 -qsimd=auto -qhot -qstrict -qprefetch`.[2] Figure 3 compares three compilation versions with this baseline. **AutoOpenMP** is an automatically parallelized version using OpenMP and SIMDization generated by *bgxlc_r* by adding the option `-qsmp=auto` to the compilation command. **AutoOpenMP+TLS** is the same code as AutoOpenMP with TLS applied by *SpecEval* to some loops that were not parallelized by bgxlc because of may dependencies. The `-qsmp` option is modified to-`qsmp=auto:speculative`. **Oracle** is obtained by applying TLS incrementally to candidate loops. If the application of TLS to a candidate loop degrades performance improvement then that loop is rejected from TLS by the Oracle. Oracle uses an heuristic that *knows* which loops are profitable for TLS and therefore is a limit study for the best performance that could be obtained using TLS in *SpecEval*. For most applications Oracle is not a practical solution, but in a performance evaluation study it provides valuable information to indicate how well the heuristic that selects loops for speculation is doing.

Figure 3 indicates that there are three classes of benchmarks - benchmarks that achieve speedup with TLS (`milc`, `lbm`, `bzip2`, `mcf`, `namd`, `hmmer`), benchmarks where performance neither improves nor degrades (`sphinx3`, `gobmk`) and benchmarks that suffer performance degradation with TLS (`h264ref`, `sjeng`). The superior oracle performance of `mcf` is explained by some loops that contain function calls that introduce dependencies. In Section 0.6.4 an heuristic that excludes these loops from TLS results in comparable performance to the oracle.

The *coverage* of a loop is the percentage of the total execution time of the program that can be attributed to that loop. There is limited performance improvement when TLS is applied to *cold* loops due to the speculative thread-creation overhead. `bzip2`, `sjeng` are benchmarks that contain speculative loops with poor coverage (Table 3).

The BG/Q transactional-memory subsystem supports two modes for speculative execution: a long-running (LR) mode and a short-running (SR) mode [3]. Only the LR mode is available for TLS at this time. In the LR mode the L1 cache must be flushed at the start of each speculative region. As a result, a significant number of L1 cache misses happens at the start of a speculative region leading to the increase in L1 cache misses observed in the experimental evaluation and limiting TLS performance. In BG/Q the hardware support for transactional memory is built on top of the hardware support for TLS — the main distintction between TLS and TM is the need to enforce the commit order in TLS. Wang *et al.* describe SR and LR in BG/Q [**?**].

As the results in Figure 4 indicate, amongst the PolyBench/C benchmarks, `cholesky` and `dynprog` suffer from the flush-effect of TLS (Table 4). Although this hypothesis cannot be experimentally evaluated at the moment, it is possible that the *Short-running* (SR) mode [3] would be more suitable for the speculative execution of these benchmarks.

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 dynamic instruction path length. `Jacobi` and `seidel` are two benchmarks that experience a significant path-length increase (Table 5) thus limiting/degrading their performance. The rest of the section explains in details the different sources of the BG/Q TLS overhead.

[2]By default the -O5 level turns on the automatic vectorization (SIMDization) and the various optimizations of hot loops (e.g. loop unrolling etc.). The option `-qstrict` maintains the correct semantics of the program when using higher-level optimizations. The option `-qprefetch` enables prefetching.

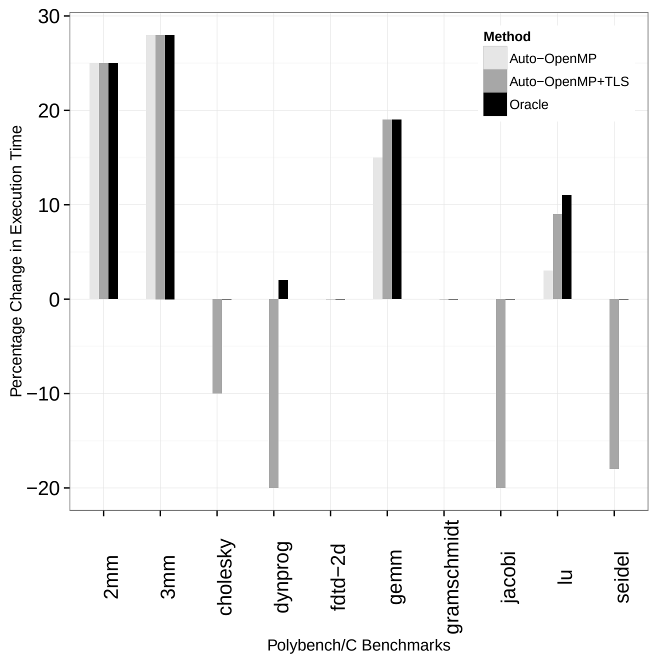

Figure 4: Percentage change in execution time from different parallelization techniques for PolyBench/C benchmarks over the auto-SIMDized code. The OpenMP and TLS versions use 4 threads.

0.6 Limitations to TLS Performance

The performance changes due to TLS performance is affected by the amount of time spent in speculated loops, by the cost of misspeculation, and by changes in cache behaviour and on the instruction path length of the programs.

0.6.1 Number of loops parallelized and their coverage

Table 3 shows the total number of loops in each benchmark, the number of loops parallelized by each of the three different parallelization techniques — automatic OpenMP parallelization, automatic SIMDization and speculatively parallelization, the coverage of speculatively parallelized loops in the SPEC2006 and PolyBench/C benchmarks, and the percentage of speculative regions that are successfully committed.

The interesting benchmark in Table 3 is `h264ref` because it has the highest number of speculatively parallelized loops with good coverage among all other benchmarks but still it experiences a slowdown due to speculative execution, as seen in Figure 3. Experiments reveal that function calls from within the speculative loop body introduce dependencies during run-time and those dependencies are not detected by the context-insensitive dependence profiling. This type of misspeculation overhead also limits the TLS performance for `sjeng`.

The performance improvement from TLS for `milc` is limited due to the speculative execution of loops with poor coverage that introduces TLS thread creation overhead. An ideal TLS candidate loop will have high coverage, ı.e. it is a hot loop, with very 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.

The PolyBench/C benchmarks `2mm` and `3mm` execute matrix multiplication. Their large loops can be parallelized with OpenMP and thus are not candidates for speculation. For `gemm`, the speculatively parallel

Table 3: Number of loops parallelized by OpenMP, SIMDized and speculated with the coverage of speculated loops. Coverage is the fraction of total time spend in the speculated loops (expressed as a percentage). Rightmost column is the percentage of speculative threads the successfully committed.

Suite	Benchmarks	Total	OpenMP	SIMDized	Speculative		%
		# Loops	# Loops	# Loops	# Loops	Cover.	Commit
SPEC2006	lbm	23	4	0	5	98 %	94 %
	h264ref	1870	179	3	47	82 %	12 %
	hmmer	851	105	17	30	80 %	79 %
	mcf	52	9	0	12	65 %	68%
	sjeng	254	9	0	16	32 %	8%
	sphinx3	609	11	0	2	91 %	29%
	bzip2	232	4	0	2	35 %	78%
	gobmk	1265	0	0	0	0 %	-
	milc	421	7	2	22	33 %	79%
	namd	619	9	7	25	92 %	80%
PolyBench/C	2mm	20	7	3	0	0 %	-
	3mm	27	10	3	0	0 %	-
	gemm	13	3	4	4	40 %	89%
	gramschmidt	10	3	0	0	0 %	-
	jacobi	9	3	0	2	3 %	78%
	lu	8	3	1	5	45 %	89%
	seidel	7	4	0	2	3 %	70%
	cholesky	9	0	0	4	10 %	79%
	dynprog	9	7	0	3	18 %	82%
	fdtd_2d	14	2	0	3	20 %	68%

loops have good coverage that accounts for the speedup from TLS. But for `cholesky`, `dynprog` and `fdtd-2d`, the poor coverage of loops results in a TLS slow-down.

`Gramschmidt` does not have any speculative loops because *cold* loops (less than 1.2% coverage) are filtered out by the speculation heuristic because they are not beneficial for TLS [31].

0.6.2 Misspeculation Overhead

The squashing of threads because of dependencies and the re-execution of the parallel section sequentially imposes overhead that results in performance degradation. In the absence of an inter-procedural data-dependence analysis, the speculation of loops with function calls is risky because the execution of the callee may introduce dependencies that were not detected by the analysis and may thus lead to thread squashing. The measurement of misspeculation overhead revealed that some of the benchmarks contain loops where function calls introduce actual dependencies during run-time.

A speculative thread either commits successfully or end in an non-committed state — in which case cache-line versions associated with the thread are discarded.

The *se_print_stats* function from the *speculation.h* header file in the BG/Q runtime system is used to collect various statistics for a speculative region including the number of successfully committed/non-committed threads.

The percentage of successfully committed threads is a good proxy measurement for the misspeculation overhead. Ideally a benchmark that can benefit from TLS should have a high percentage of successful completion of speculative threads.

Table 3 also shows 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 introduce new dependencies during run-time. Though `h264ref` has a high number of loops speculatively parallelized (Table 3), 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 an increase in L1 cache misses, likely caused by the LR-mode cache flush, while `jacobi` and `seidel` suffer from a significant increase in dynamic instruction path length. These two benchmarks contain loops that have low iteration counts.

Two approaches can be used to overcome the misspeculation overhead due to function calls:

- **Conservative Approach:** The compiler conservatively does not allow loops with function calls to be executed in parallel regardless of whether the function call changes the dependence behaviour of the loop. This heuristic might miss parallelization opportunities where the function call is harmless.

- **Precise Approach:** A more sophisticated inter-procedural dependence analysis technique is used to decide whether the function call introduces new dependencies during run-time.

The performance effect of adopting the conservative approach on the SPEC2006 benchmarks is evaluated in the following section.

0.6.3 TLS Startup Overhead on the BG/Q

Starting a TLS region on the BG/Q requires operating system actions, primarily to setup threads with a special virtual-memory structure. Knowing the cost of the setup process is important to understanding the profitability of speculating the execution of a loop. Profitable use of TLS generally requires choosing an speculative region of code that is large enough to amortize these overheads yet small enough to have a low conflict probability. A separate set of experiments measure the TLS startup overhead. In these experiments, a simple loop with independent iterations is executed many times: sometimes sequentially, sometimes in parallel using OpenMP, and sometimes speculatively in parallel using TLS. The test was constructed so that the iteration count and the iteration independence could not be determined at compile time. Furthermore, the ordering of loop executions using OpenMP and TLS were permuted to search for any 'switching penalty' between the two methods, but none was found. Cycle counts for each loop execution were obtained from the processor's time-base register. The serial loop executed in the range of 174 to 228 cycles. As shown in Figure 6, the two parallel methods, while similar to each other, exhibit a complex distribution of execution cycle counts. We were unable to determine the cause of the distribution's complexity, but one thing is clear: starting an OpenMP or TLS region on the BG/Q takes many hundreds of thousands of cycles, and sometimes nearly an order of magnitude more. Even though the timings are highly variable, they were also deterministic. This relatively large overhead associated with TLS region startup explains the success of heuristics, such as those used in this work, that limit the use of TLS to high-coverage loops.

0.6.4 Preventing Speculation of Loops with Side-Effect Function Calls

Allowing speculative execution of loops with function calls for `h264ref` and `sjeng` introduces misspeculation overhead that results in performance degradation. Function calls inside speculated loop bodies in these benchmarks introduce dependencies across iterations at run time. The static dependence analysis in LLVM is currently unable to detect these dependencies. An inter-procedural dependence analysis based on profiling information is not yet available.

This section studies the performance effect of preventing the speculative execution of loops with function calls that may have side effects.

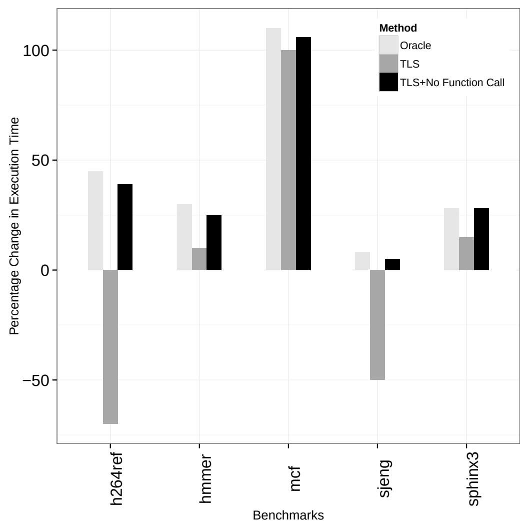

Figure 5: Percentage change in execution time of SPEC2006 benchmarks over auto-SIMDized code after filtering speculative execution of loops with function calls.

The results in Figure 5 indicate that preventing speculative execution of loops with function calls changes the performance degradation of `h264ref` and `sjeng` into performance gains. The percentage of successfully committed threads jumps up from 12% to 96% for `h264ref` and from 8% to 97% for `sjeng`. Using this approach, a 32% change is achieved for `hmmer` because `hmmer` contains some loops where new dependencies are introduced at run-time due to function calls. Also the performance of `mcf` comes close to the performance of the oracle version. Performance *does not* degrade for any of these benchmarks, thus indicating that there are no loops in these benchmarks that are both hot and that have function calls that affect the run-time dependencies.

The performance of the PolyBench/C benchmarks does not change when this approach is used. Most PolyBench/C benchmarks are kernel benchmarks and the loops inside them do not contain function calls.

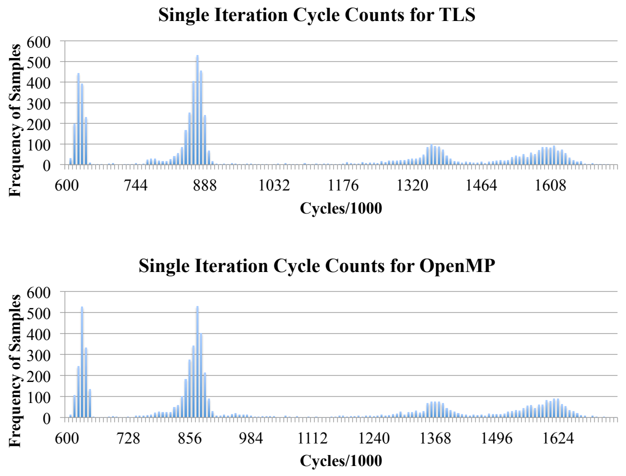

Figure 6: Distribution of the number of cycles required to execute a single iteration of a simple loop with TLS and OpenMP.

0.6.5 L1 Cache Miss Rate

One of the most dominant run-time overhead in the BG/Q TLS is caused by the loss of L1 cache support due to the L1 flush 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 the BG/Q is used to collect the L1 miss statistics. Table 4 gives the L1 cache hit rate for the sequential version and the three parallel versions of the SPEC2006 and PolyBench/C benchmarks.

Table 4: 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 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 the TLS region in the LR mode. Apart from the function calls that introduce data-dependencies, 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 3, 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.[3] 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.

[3]The instruction path length is the total number of instructions executed by an application.

0.6.6 Increase in Instruction Path Length

Automatic OpenMP and speculative parallelization insert calls to OpenMP and TLS run-time functions, respectively, into 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. Table 5 shows the effect of TLS on code growth.

Table 5: 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 %

The code growth for the PolyBench/C benchmarks is higher than for the SPEC2006 benchmarks because loops constitute a major portion of the PolyBench/C benchmarks and therefore the parallelization of these loops affects the code size more significantly. For SPEC2006 benchmarks the code growth is relatively smaller, the highest being for the speculative parallelization of `h264ref` where a large number of loops are speculatively parallelized (Table 3).

As seen in Table 5, `jacobi` and `seidel` experience a significant 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 3). Loops whose speculation leads to non-trivial code growth should be only judiciously speculated.

0.7 Performance Trends

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 7.

As seen in the figure, `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 [29]. The performance of `milc` scales up to 8 threads. For `hmmer` and `sjeng`, the performance improvement for TLS is negligible in

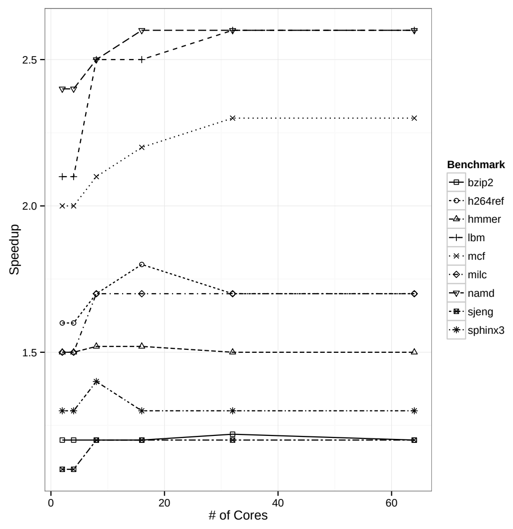

Figure 7: Scalability of speculatively parallelized versions of the SPEC2006 benchmarks.

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. There is also very little performance change with increasing number of threads.

0.7.1 Using Clauses with the Basic TLS Pragma

All the experiments discussed so far use the basic TLS pragma available for TLS on the BG/Q. But the *bgxlc_r* compiler offers many OpenMP-like clauses that can be used to optimize the performance of the speculative loop [32]. 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 of threads and distribution of work among threads.

This case study illustrates the use of specific clauses on the `lbm` and `h264ref` benchmarks. These benchmarks are chosen because `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 their performance effects. This manual instrumentation allowed the parallelization of more loops.

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 the scoping of variables results in negligible performance variations for these two benchmarks (improvements of .05 % and .01 % respectively for `lbm` and `h264ref`), and that 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 [4, 33]. One approach for automatic modification of the pragmas for work sharing is to use machine-learning techniques [4, 33]. 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 [34].

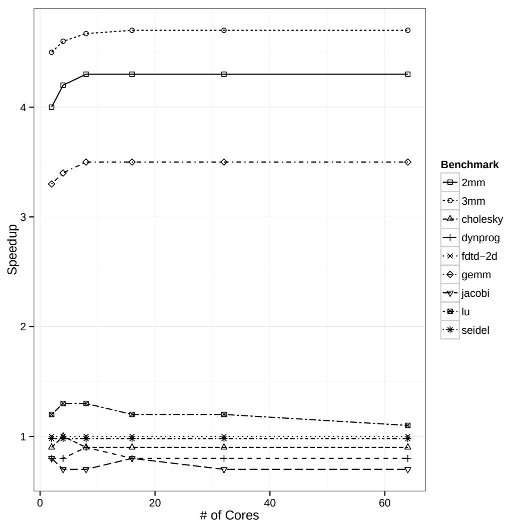

Figure 8: Scalability of speculatively parallelized versions of the PolyBench/C benchmarks.

0.8 Lessons Learned

The introduction posed some important research questions regarding the use of TLS to improve performance. This study is limited to standard benchmarks that were not designed with the idea of exploiting TLS in mind and is limited to the first implementation of TLS commercial available. Nonetheless, it has indicated that obtaining significant performance gains from TLS alone may be harder than previously thought. The overall performance gains reported appear underwhelming and would discourage the use of TLS if significant effort would be required by the final user. However, the good news is that, given that the machinery to support TLS is already implemented in the hardware, once the compilation and run-time system support is in place, deploying TLS to an application requires minimum effort. The confirmation that for the majority of the applications the materialization of may-dependences is independent of the program input — and therefore the realization that a single profiling run would be sufficient to determine which may-dependences can be safely speculated — makes the use of TLS even simpler.

A second lesson learned is that, even though the cost for starting a TLS region is similar to the cost of starting an OpenMP region, both of them can be high. An interesting direction for future improvement, which requires access to the IBM proprietary software stack, would be to identify which portion of this overhead is due to the hardware and to the software and to try to improve the software stack to reduce this cost.

0.9 Conclusion

This research connected the dependence analysis in one compiler (LLVM) and the support for TLS in another (*bgxlc_r*) to create a new automatic speculative parallelization framework, *SpecEval*, that allowed for an early evaluation of the performance effects of TLS on the BlueGene/Q supercomputer. This evaluation found that many factors must be taken into consideration when selecting candidate loops for speculation. These factors include: number and coverage of speculative loops, misspeculation overhead due to function calls inside the loop body, increase in L1 cache misses and dynamic instruction path length increase.

The evaluation, using benchmarks from the SPEC2006 and PolyBench/C suites, found that benchmarks that have loops with good coverage and without dependence violation, such as `lbm`, are well suited for TLS. But the widespread use of TLS in benchmarks that contain function calls with side effects will require more sophisticated inter-procedural dependence analysis in the compiler.

This research was originally motivated by the idea that information obtained from profiling is expected to vary according to the program input used for profiling [35]. Thus, it was pleasant to confirm earlier claims that, in practice, loops' dependence behaviour does not change based on inputs on a wide range of benchmarks. Though we presented the performance study of a benchmark (2d-hull) that has loops with varied dependence behaviour across inputs, but existence of similar applications is not known to us. Based on this finding, single input data-dependence profiles could be used to find the speculation candidate loops in this study. Also the study in this report focused ion the SPEC2006 and PolyBench/C benchmarks mainly because both of these benchmarks have been used extensively in previous studies of TLS. Our investigation confirmed that variations of may-dependence do not occur in either suite. This is an important finding and we argue that TLS is necessary to ensure that the speculation used by the compiler is safe.

Acknowledgments.

We thank Paul Berube and Matthew Gaudet for providing valuable feedback throughout the course of this research. I would also like to thank Jeff Hammond at the Argonne National Laboratory for his support and for giving access to the BG/Q. This research was partially funded by the Natural Science and Engineering Research Council of Canada through a Collaborative Research and Development grant and through the IBM Centre for Advanced Studies.

Bibliography

[1] T. Chen, J. Lin, X. Dai, W.-C. Hsu, and P.-C. Yew, "Data dependence profiling for speculative optimizations," in *Compiler Construction*, ser. Lecture Notes in Computer Science, E. Duesterwald, Ed. Springer Berlin Heidelberg, 2004, vol. 2985, pp. 57–72.

[2] J. Lin, T. Chen, W.-C. Hsu, P.-C. Yew, R. D.-C. Ju, T.-F. Ngai, and S. Chan, "A compiler framework for speculative analysis and optimizations," in *Programming Language Design and Implementation (PLDI)*, 2003, pp. 289–299.

[3] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, "Evaluation of blue gene/q hardware support for transactional memories," in *International Conference on Parallel Architectures and Compilation Techniques(PACT)*, 2012, pp. 127–136.

[4] G. Tournavitis, Z. Wang, B. Franke, and M. F. 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)*, 2009, pp. 177–187.

[5] H. Kim, N. P. Johnson, J. W. Lee, S. A. Mahlke, and D. I. August, "Automatic speculative doall for clusters," in *Code Generation and Optimization (CGO)*, 2012, pp. 94–103.

[6] K.-F. Faxen, K. Popov, L. Albertsson, and S. Janson, "Embla - data dependence profiling for parallel programming," in *Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008. International Conference on*, 2008, pp. 780–785.

[7] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *International Symposium on Computer Architecture (ISCA)*, 1995, pp. 414–425.

[8] M. Franklin and G. S. Sohi, "Arb: A hardware mechanism for dynamic reordering of memory references," *IEEE Trans. Comput.*, vol. 45, no. 5, pp. 552–571, May 1996.

[9] M. Cintra and J. Torrellas, "Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors," in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, ser. HPCA '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 43–. [Online]. Available: http://dl.acm.org/citation.cfm?id=874076.876479

[10] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VIII, 1998, pp. 58–69.

[11] V. Krishnan and J. Torrellas, "The need for fast communication in hardware-based speculative chip multiprocessors," in *International Conference on Parallel Architectures and Compilation Techniques(PACT)*, 1999, pp. 24–.

[12] P. Marcuello and A. González, "Clustered speculative multithreaded processors," in *International Conference on Supercomputing (ICS)*, 1999, pp. 365–372.

[13] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," in *International Symposium on Computer Architecture (ISCA)*, 2000, pp. 1–12.

[14] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi, "Speculative versioning cache," in *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, ser. HPCA '98, 1998, pp. 195–.

[15] F. Dang, H. Yu, and L. Rauchwerger, "The r-lrpd test: speculative parallelization of partially parallel loops," in *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, April 2002, pp. 10 pp–.

[16] P. Rundberg and P. Stenstrom, "An all-software thread-level data dependence speculation system for multiprocessors," *Journal of Instruction-Level Parallelism*, vol. 3, p. 2002, 2001.

[17] C. E. Oancea and A. Mycroft, "Software thread-level speculation: An optimistic library implementation," in *Proceedings of the 1st International Workshop on Multicore Software Engineering*, ser. IWMSE '08, 2008, pp. 23–32.

[18] M. Cintra and D. R. Llanos, "Toward efficient and robust software speculative parallelization on multiprocessors," in *Principles and practice of parallel programming(PPoPP)*, 2003, pp. 13–24.

[19] J. Da Silva and J. G. Steffan, "A probabilistic pointer analysis for speculative optimizations," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 416–425.

[20] L.-L. Chen and Y. Wu, "Aggressive compiler optimization and parallelization with thread-level speculation," in *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, 2003, pp. 607–614.

[21] X. Dai, A. Zhai, W.-C. Hsu, and P.-C. Yew, "A general compiler framework for speculative optimizations using data speculative code motion," in *Code Generation and Optimization (CGO)*, 2005, pp. 280–290.

[22] M. Kim, H. Kim, and C.-K. Luk, "Sd3: A scalable approach to dynamic data-dependence profiling," in *International Symposium on Microarchitecture(MICRO)*, 2010, pp. 535–546.

[23] P. Wu, A. Kejariwal, and C. Caşcaval, "Languages and compilers for parallel computing," 2008, ch. Compiler-Driven Dependence Profiling to Guide Program Parallelization, pp. 232–248.

[24] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas, "Posh: A tls compiler that exploits program structure," in *Principles and practice of parallel programming(PPoPP)*, 2006, pp. 158–167.

[25] N. Ioannou, J. Singer, S. Khan, P. Xekalakis, P. Yiapanis, A. Pocock, G. Brown, M. Lujan, I. Watson, and M. Cintra, "Toward a more accurate understanding of the limits of the tls execution paradigm," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, 2010, pp. 1–12.

[26] N. P. Johnson, H. Kim, P. Prabhu, A. Zaks, and D. I. August, "Speculative separation for privatization and reductions," in *Programming Language Design and Implementation (PLDI)*, 2012, pp. 359–370.

[27] K. L. Clarkson, K. Mehlhorn, and R. Seidel, "Four results on randomized incremental constructions," *Computational Geometry*, vol. 3, no. 4, pp. 185 – 212, 1993.

[28] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Code Generation and Optimization (CGO)*, Palo Alto, California, Mar 2004.

[29] V. Packirisamy, A. Zhai, W.-C. Hsu, P.-C. Yew, and T.-F. Ngai, "Exploring speculative parallelism in spec2006," in *International Symposium on Performance Analysis of Systems and Software(ISPASS)*, 2009, pp. 77–88.

[30] M. F. Spear, K. Kelsey, T. Bai, L. Dalessandro, M. L. Scott, C. Ding, and P. Wu, "Fastpath speculative parallelization," in *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC'09, 2010, pp. 338–352.

[31] A. Bhattacharyya and J. N. Amaral, "Automatic speculative parallelization of loops using polyhedral dependence analysis," in *Proceedings of the First International Workshop on Code OptimiSation for MultI and Many Cores*, ser. COSMIC '13, 2013, pp. 1:1–1:9.

[32] "Ibm xl c/c++ for blue gene/q, v12.1," http://www-03.ibm.com/software/products/us/en/xlcc+forbluegene/.

[33] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: A machine learning based approach," in *Principles and practice of parallel programming(PPoPP)*, 2009, pp. 75–84.

[34] Y. Lin, C. Terboven, D. Mey, and N. Copty, "Automatic scoping of variables in parallel regions of an openmp program," in *Shared Memory Parallel Programming with Open MP*, ser. Lecture Notes in Computer Science, 2005, vol. 3349, pp. 83–97.

[35] P. Berube and J. Amaral, "Combined profiling: A methodology to capture varied program behavior across multiple inputs," in *International Symposium on Performance Analysis of Systems and Software(ISPASS)*, 2012, pp. 210–220.