Data-dependence Profiling to Enable Safe Thread Level Speculation

Arnamoy Bhattacharyya  
University of Alberta  
Edmonton, Alberta  
Canada  
arnamoy@ualberta.ca

José Nelson Amaral  
University of Alberta  
Edmonton, Alberta  
Canada  
jamaral@ualberta.ca

Hal Finkel  
Argonne National Laboratory  
hfinkel@anl.gov

ABSTRACT

Data-dependence profiling is a technique that enables a compiler to judiciously decide whether the execution of a loop — which the compiler could not prove to be dependence free — should be speculated through the use of Thread Level Speculation (TLS). The data collected by a data-dependence profiler can be used to predict if may dependencies reported by a compiler static analysis are likely to materialize at runtime. A cost analysis can then be used to decide that some loops with a lower probability of dependence should be speculatively parallelized. This paper addresses 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 paper describes SpecEval, an automatic speculative parallelization framework that uses single-input data-dependence profiles to find speculation candidates in the SPEC2006 and PolyBench/C benchmarks. This paper also presents a performance evaluation of TLS implementation in IBM’s BlueGene/Q supercomputer and shows that the performance of TLS is affected by several factors, including the number of speculated loops, the execution-time coverage of speculated loops, the miss-speculation overhead, the L1 cache miss rate and the effect on dynamic instruction path length.

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 compiler 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.

If a compiler cannot determine whether there will be a dependence at run-time, data-dependence profiling can be used to predict the actual occurrence of dependencies at runtime [5, 20]. Data-dependence profiling records the memory addresses accessed by possibly dependent load/store instructions for all may-dependencies. These 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 provides a safe fall-back execution path for the runs in which a profile-based prediction of absence of dependence turns 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 compiler-generated code and runtime functionality. Hardware-supported thread-level speculation (TLS) offers an easier-to-implement and more efficient fall-back path [33].

Thus, two important questions 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? Earlier studies had suggested that, for widely used sets of benchmarks, the dependence behaviour does not change with the data input [10, 15, 31]. This result is confirmed by our own study, described in this paper, 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. A more recent study has identified benchmarks where the dependence behaviour does vary with changes to the data input to the benchmark [32]. This same study reveals that often changes to dependence behaviour is present but rare, thus strengthening the argument for the use of TLS as performance-favourable choice. Earlier proposals to enable speculative execution of loop iterations based on data-dependency invariability did not provide a safety net to ensure that all possible runs of the program would produce the correct result [31]. One of the main contributions of this paper is to combine this simpler data-dependence profiling
prediction with TLS to provide a framework for speculative parallelization.

Using a single 16-core compute node of the TLS-enabled IBM’s BlueGene/Q supercomputer as an experimental platform, this paper makes the following contributions:

- 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.

2. RELATED WORK

Architecture techniques to support TLS have been extensively studied and found to be successful due to their lower overhead. Multiscalar [28] introduced hardware-based TLS and initially used hardware buffers called Address Resolution Buffers (ARB) [11]. Later studies relied on shared-memory cache coherence protocols to support TLS [7, 12, 13, 17, 23, 30]. Proposals on software-only TLS revealed very high overhead for such systems [25, 6].

Several profiling and speculation frameworks have been proposed [4, 5, 8, 9, 16, 20]. Embila is a data-dependence profiler that supports TLS [10]. Chen et al. propose a data-dependence profiler developed for speculative optimizations [5]. 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 [37]. 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 [22].

Previous research found that there is limited variability in the dependence behaviour of loops across inputs [31, 15, 10]. 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.

The IBM POWER8 processor introduces transactions with suspended regions. Using this mechanism to implement TLS, Le et al. report significant performance gain in a micro benchmark [19]. However, Odaira and Nakaike found it much more difficult to produce significant performance improvements when implementing TLS on top of off-the-shelf hardware-supported transactional memory [26]. More recently, by manually changing loops in two SPEC CPU benchmarks, Nakaike et al. report speedups of 15% and 25% due to TLS supported by the suspend/resume instructions [24].

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. This study examined loops from 57 different benchmarks from the SPEC2006, PolyBench/C, Biobenchmarks and NAS benchmark suites with different inputs as listed in Table 1. The study confirms the observation of previous studies [10, 15, 31], and reveals that none of these benchmarks have a loop where variations on dependence behaviour across inputs was observed — admittedly the study used a small number of inputs for each benchmark. For loops that were signalised as containing may-dependencies by the compiler, either the dependence materializes for all program inputs examined 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 the potential use of TLS because it indicates that costly many-input profiling is seldom required. Furthermore, von Koch and Franke found that dependences that inhibit loop parallelization are fairly stable across program inputs [32].

4. SPECEVAL: A SPECCULATIVE PARALLELIZATION FRAMEWORK

SpecEval is 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 1, 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 can be collected. The debug information is used to map executable code to source code and thus 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
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 the for loops that are found to be speculation candidates based on a heuristic. The heuristic selects a loop for speculation if the may dependences 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 to signal to the XL compiler that the loop should be speculated. This BG/Q-specific pragma divides the total iteration space into chunks of:

\[
\frac{\text{number of iterations}}{\text{number of threads}}
\]

Figure 2 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. A different input is used in the profiling run than the input used for the test run.

```c
#pragma speculative for
for ( i = 0; i < x; i++ )
{
       //loop body
}
```

Figure 2: A Sample loop with the TLS pragma inserted

This framework leveraged the dependence analysis and the versatility of the LLVM framework [18] 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.

5. PROFILE-DRIVEN TLS

This section evaluates the impact of applying profile-driven TLS to benchmarks from the SPEC2006 and PolyBench/C benchmark suites using SpecEval — these suites have been used in previous TLS studies [15, 27, 37]. 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 in the execution time.

Previous studies have used TLS for benchmarks with many parallel loops without regards for the nature of potential dependencies [25, 29]. 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

1The `r` option generates thread-safe code.
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.

5.1 Interaction between TLS, AutoSIMD, and AutoOpenMP parallelizers

A loop is SIMDizable if its execution can use the vector instructions and vector registers existing in the machine. Elsewhere such loops are also called vectorizable. 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 -qbot -qstrict -qprefetch. 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 -qomp=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. Loops that are parallelized with the OpenMP option are called parallelizable elsewhere. The -qomp option is modified to -qomp=auto:speculative. Oracle is obtained by applying TLS incrementally to candidate loops. If the application of TLS to a candidate loop degrades performance then that loop is rejected from TLS by the oracle. The oracle uses an unrealizable heuristic that knows which loops are profitable for TLS. Therefore the experiment with the oracle is a limit study that estimates the best performance that could be obtained using TLS in SpecEval. The oracle is not a practical solution for actual application programs, but in a performance evaluation study it provides valuable information to indicate how well the heuristic that selects loops for speculation is doing.

The results in Figure 3 indicate that there are three classes of benchmarks - benchmarks that achieve speedup with TLS (milc, lpm, 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 for mcf is explained by some loops, speculated by SpecEval, that contain function calls that introduce dependencies. In Section 6.4 a heuristic that excludes these loops from TLS gives comparable performance to that of 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 2).

The BG/Q transactional-memory subsystem supports two modes for speculative execution: a long-running (LR) mode and a short-running (SR) mode [33, 34, 35]. Only the LR mode was available for TLS during this study. 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 distinction between TLS and TM is the need to enforce the commit order in TLS. Wang et al. describes the SR and LR modes of BG/Q [34].

As the results in Figure 4 indicate, amongst the Polybench/C benchmarks, the performance of cholesky and dynprog is affected by the flushing of the L1 cache required for TLS in BG/Q (Table 3). Although this hypothesis could not be experimentally evaluated in this study, it is possible that the use of the Short-running (SR) mode [33] would be more suitable for the speculative execution of these benchmarks.

The starting of a TLS execution requires the execution of additional code — such as saving of register context before entering a speculative region and obtaining a speculative ID — that accounts for some of 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 4) thus limiting/degrading their performance.

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.
6. LIMITATIONS TO TLS PERFORMANCE

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

6.1 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. With rare exception, the serial loop executed in 217 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.

6.2 Loops parallelized and their coverage

Table 2 shows the total number of loops in each of the studied benchmarks, the number of loops parallelized by each of the three different parallelization techniques — automatic OpenMP parallelization, automatic SIMDization and speculative parallelization via TLS. The table also presents 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 2 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, i.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 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 as they are not beneficial for TLS [3].

6.3 Misspeculation Overhead

The abortion of speculative execution 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 a 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 2 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 2), the presence of dependence resulted from the function calls inside the loops accounts for the slowdown. Mcf, hammer, 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 signifi-
Table 2: Number of loops parallelized by OpenMP, SIMDized and speculated with the coverage of speculated loops. Coverage is the fraction of total time spent in the speculated loops (expressed as a percentage). Rightmost column is the percentage of speculative threads that successfully committed.

<table>
<thead>
<tr>
<th>Suite</th>
<th>Benchmarks</th>
<th>Total # Loops</th>
<th>OpenMP Speculative # Loops</th>
<th>SIMDized Speculative # Loops</th>
<th>Speculative Cover.</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPEC2006</td>
<td>ibm</td>
<td>23</td>
<td>4</td>
<td>0</td>
<td>98 %</td>
<td>94 %</td>
</tr>
<tr>
<td></td>
<td>h264ref</td>
<td>1870</td>
<td>179</td>
<td>3</td>
<td>47 %</td>
<td>82 %</td>
</tr>
<tr>
<td></td>
<td>hmer</td>
<td>851</td>
<td>105</td>
<td>17</td>
<td>30 %</td>
<td>80 %</td>
</tr>
<tr>
<td></td>
<td>mcf</td>
<td>52</td>
<td>9</td>
<td>0</td>
<td>12 %</td>
<td>65 %</td>
</tr>
<tr>
<td></td>
<td>sjeng</td>
<td>254</td>
<td>9</td>
<td>0</td>
<td>16 %</td>
<td>32 %</td>
</tr>
<tr>
<td></td>
<td>sphinx3</td>
<td>609</td>
<td>11</td>
<td>0</td>
<td>2 %</td>
<td>91 %</td>
</tr>
<tr>
<td></td>
<td>bzip2</td>
<td>232</td>
<td>4</td>
<td>0</td>
<td>2 %</td>
<td>35 %</td>
</tr>
<tr>
<td></td>
<td>gobmk</td>
<td>1265</td>
<td>0</td>
<td>0</td>
<td>0 %</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>milc</td>
<td>421</td>
<td>7</td>
<td>2</td>
<td>22 %</td>
<td>33 %</td>
</tr>
<tr>
<td></td>
<td>namd</td>
<td>619</td>
<td>9</td>
<td>7</td>
<td>25 %</td>
<td>92 %</td>
</tr>
<tr>
<td>PolyBench/C</td>
<td>2mm</td>
<td>20</td>
<td>7</td>
<td>3</td>
<td>0 %</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>3mm</td>
<td>27</td>
<td>10</td>
<td>3</td>
<td>0 %</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>gemm</td>
<td>13</td>
<td>3</td>
<td>4</td>
<td>40 %</td>
<td>89 %</td>
</tr>
<tr>
<td></td>
<td>gramoschmidt</td>
<td>10</td>
<td>3</td>
<td>2</td>
<td>0 %</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>jacobia</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>3 %</td>
<td>78 %</td>
</tr>
<tr>
<td></td>
<td>lu</td>
<td>8</td>
<td>3</td>
<td>1</td>
<td>5 %</td>
<td>45 %</td>
</tr>
<tr>
<td></td>
<td>seidel</td>
<td>7</td>
<td>4</td>
<td>0</td>
<td>2 %</td>
<td>3 %</td>
</tr>
<tr>
<td></td>
<td>cholesky</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>4 %</td>
<td>10 %</td>
</tr>
<tr>
<td></td>
<td>dynprog</td>
<td>9</td>
<td>7</td>
<td>0</td>
<td>3 %</td>
<td>18 %</td>
</tr>
<tr>
<td></td>
<td>stdl2d</td>
<td>14</td>
<td>2</td>
<td>0</td>
<td>3 %</td>
<td>20 %</td>
</tr>
</tbody>
</table>

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

The performance effect of adopting the conservative approach on the SPEC2006 benchmarks is evaluated in Section 6.4.

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.

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 hmer because hmer 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.

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. 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 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 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. These two benchmarks have a high percentage of successful completion of speculative threads — see Table 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 is likely the cause 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 ftdt-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.

### 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 4 shows the effect of TLS on code growth.

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 2).

The data in Table 4 indicate that jacobi and seidel experience a significant code growth that is likely to explain their slowdown. Benchmarks such as cholesky, dynprog and ftdt-2d also suffer code growth due to the presence of loops with poor coverage (see Table 2). Loops whose speculation leads to non-trivial code growth should be only judiciously speculated.

### 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. This section examines the scalability of TLS performance 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.
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 [27]. 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. There is also very little performance change with increasing number of threads.

### 7.1 Using Clauses with the Basic TLSPragma

All the experiments discussed so far use the basic TLS pragma available for TLS on BG/Q. The bgxlc_r compiler also supports OpenMP-like clauses that can be used to optimize the performance of speculative loops [14]. 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 the number of threads and the 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. The 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 mention that finding the best-suited (OpenMP) pragma automatically in loops is non-trivial and needs programmer’s support [31, 36]. One approach for automatic modification of the pragmas for work sharing is to use machine-learning techniques [31, 36]. 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 [21].

### 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 hardware-supported TLS commercially available. Nonetheless, the results presented indicate 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 application the materialization of may-dependencies is independent of the program input — and therefore the solution that a single profiling run would be sufficient to determine which may-dependencies 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.

### 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 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, mis speculation 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 blm, 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 [1]. While for some programs where variation in dependence behaviour with input does occur [2], it was pleasant to confirm earlier claims that, in practice, for many benchmarks loops’ dependence behaviour does not change based on inputs on a wide range of benchmarks. Based on this finding, single input data-dependence profiles could be used to find the speculation candidate loops in this study.

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. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

10. REFERENCES


