Exploring Speculative Parallelism in SPEC2006

Venkatesan Packirisamy, Antonia Zhai, Wei-Chung Hsu, Pen-Chung Yew and Tin-Fook Ngai†
University of Minnesota, Minneapolis. *†Intel Corporation*
{packve,zhai,hsu,yew}@cs.umn.edu *tin-fook.ngai@intel.com*

***Abstract*—**
The computer industry has adopted multi-threaded and multi-core architectures as the clock rate increase stalled in early 2000's. It was hoped that the continuous improvement of single-program performance could be achieved through these architectures. However, traditional parallelizing compilers often fail to effectively parallelize general-purpose applications which typically have complex control flow and excessive pointer usage. Recently hardware techniques such as Transactional Memory (TM) and Thread-Level Speculation (TLS) have been proposed to simplify the task of parallelization by using speculative threads. Potential of speculative parallelism in general-purpose applications like SPEC CPU 2000 have been well studied and shown to be moderately successful. Preliminary work examining the potential parallelism in SPEC2006 deployed parallel threads with a restrictive TLS execution model and limited compiler support, and thus only showed limited performance potential. In this paper, we first analyze the cross-iteration dependence behavior of SPEC 2006 benchmarks and show that more parallelism potential is available in SPEC 2006 benchmarks, comparing to SPEC2000. We further use a state-of-the-art profile-driven TLS compiler to identify loops that can be speculatively parallelized. Overall, we found that with optimal loop selection we can potentially achieve an average speedup of 60% on four cores over what could be achieved by a traditional parallelizing compiler such as Intel's ICC compiler. We also found that an additional 11% improvement can be potentially obtained on selected benchmarks using 8 cores when we extend TLS on multiple loop levels as opposed to restricting to a single loop level.

I. Introduction

With the advent of multi-threaded (e.g. simultaneous multi-threading (SMT) [1], hyper-threading [2]) and/or multi-core (e.g. chip multiprocessors (CMP) [3], [4]) architectures, now the challenge is to utilize these architectures to improve performance of general-purpose applications. Automatic compiler parallelization techniques have been developed and found to be useful for many scientific applications that are floating-point intensive. However, when applied to general-purpose integer-intensive applications that have complex control flow and excessive pointer accesses, traditional parallelization techniques become quite ineffective, as they need to conservatively ensure program correctness by synchronizing all potential dependences in the program. This often requires a programmer to explicitly create parallel threads and insert synchronizations. This approach is often error prone and puts a huge burden on the programmer.

There have been numerous studies on hardware support for speculative threads, which intend to ease the creation of parallel threads for programmers and compilers. Recently, Hardware Transactional Memory (HTM) has been proposed to aid the development of parallel programs; Thread-Level Speculation (TLS) has been used to exploit parallelism in sequential applications that are difficult to parallelize using traditional parallelization techniques. For example, a loop that contains an inter-thread data dependence due to loads and stores through pointers cannot be parallelized using traditional compilers; but with the help of TLS, the compiler can parallelize this loop speculatively and relying on the underlying hardware to detect and enforce inter-thread data dependences at run-time. [5], [6], [7], [8], [9], [10]

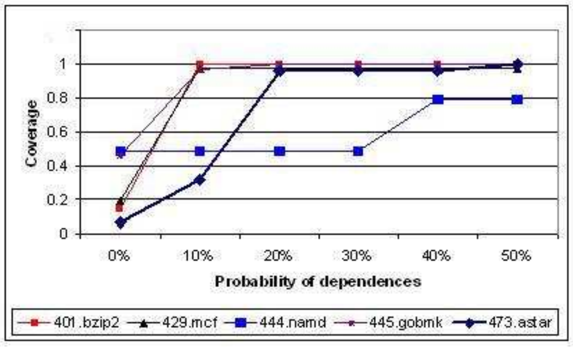


Fig. 1. Coverage obtained by parallelizing loops with certain probability of data dependences.

Though TLS has been extensively studied in the past, it is not clear how much TLS could benefit more recent benchmarks such as SPEC 2006 [11], which represent a different class of applications. Some recent studies [12] on SPEC 2006 benchmarks have shown very limited potential for TLS (less than 1%) under very conservative assumptions. In this paper, we re-examine some of these issues and give a more realistic assessment of TLS on these benchmarks using a state-of-the-art TLS compiler. By comparing the data dependence behaviors of SPEC 2000 and SPEC 2006, we show more potential parallelism in SPEC 2006 than in SPEC 2000.

One of the key detriments in parallelizing loops is the presence of cross-iteration data dependences. Figure 1 shows the results of a potential study: the percentage of execution that can potentially be parallelized if infrequently occurring cross-iteration data dependences can be *magically* resolved. The *x*-axis indicates the dependence frequency; and the *y*-axis indicates the percentage of total execution that can be parallelized. A data point at location (C, p) indicates: if loops containing only memory-resident value data dependences that occur in less than p% of threads can be parallelized, then C% of total execution can be parallelized. We can see that for these SPEC 2006 benchmarks, there are many loops with low probability data dependences. For example in 473.astar if we ignore dependences that only occur in less than 20% of all iterations, we can parallelize loops that correspond to 96% of total execution. With a traditional compiler, all these dependences would be synchronized, and thus the resulting program will exhibit poor parallel execution performance. With TLS, many of these loops could potentially be parallelized by speculating on such low probability data dependences.

Our study differs in previous studies on several aspects, and thus we believe that our results are able to accurately identify more potential for TLS than those studies. Kejariwal *et. al* [12] did not take into account the effect of compiler optimizations that could improve the performance of TLS, while previous studies [10], [7], [8], [13] have shown that compiler-based loop selection and optimizations, such as code scheduling, can significantly improve the efficiency of TLS. Furthermore, Kejariwal *et. al* [12] only considered innermost loops for TLS. In this paper, our study is not limited to a particular loop level, rather we attempt to parallelize all loops that can potentially benefit from TLS. More importantly, instead of a high-level study on performance potential of TLS, we use a state-of-the-art TLS compiler to parallelize TLS loops and study their performance using a detailed simulation infrastructure. Our results show that, with TLS-oriented compiler optimizations and optimal selection of loops, we could achieve an average of about 60% speedup for SPEC 2006 benchmarks over what could be achieved by a traditional parallelizing compiler such as Intel's ICC compiler.

As the current trend is to support more cores on a single chip, we also study the potential of enhancing TLS performance by extracting speculative threads at more than one loop level. We use a compiler-based static loop allocation scheme to efficiently schedule speculative threads from multiple loop levels. We show that an additional 11% improvement could potentially be obtained on selected benchmarks by extending TLS for multiple loop levels using eight core.

In summary, the contributions of this paper are

1) We present a detailed analysis of cross-iteration data dependences (both register- and memory-based data dependences) in SPEC 2006 benchmarks. We classify the benchmarks according to their data dependence behavior.
2) We present a comparison of cross-iteration dependence pattern of SPEC 2006 benchmarks with the SPEC 2000 benchmarks and show that the SPEC 2006 benchmarks have more potential for parallelism than the benchmarks in SPEC 2000.
3) With a state-of-art TLS compiler, we extract speculative threads from SPEC 2006 benchmarks and demonstrate that there exists additional realizable performance over a traditional parallelizing compiler.
4) We use a novel static loop allocation algorithm to study the performance potential of TLS when applied to nested loops.

The rest of the paper is organized as follows: Section II describes the related work; Section III analyzes the cross-iteration dependences that occur in SPEC 2006 benchmarks; Section IV describes our compiler framework and the evaluation methodology; Section V shows the performance of TLS and the scalability of TLS performance. In Section VI we use single -level TLS performance to study the performance potential for multi-level TLS and in Section VII we present our conclusions.

II. RELATED WORK

There has been a large body of research work on architectural design and compiler techniques for TLS [5], [6], [7], [8], [9], [10]. But all of these papers based their studies on SPEC 2000 or other older benchmarks, rather than the more recent SPEC 2006 benchmarks. The SPEC 2006 benchmarks represent a newer class of applications and it is important to examine whether the conclusions drawn for SPEC 2000 will hold for these applications. In this paper we address this issue by conducting a detailed study of SPEC 2006 benchmarks using a state-of-the-art TLS compiler.

Oplinger *et. al* [14] presented a study on the limits of TLS performance on some SPECint95 benchmarks. The impact of compiler optimizations and the TLS overhead were not taken into account in that study. Similarly, Warg *et. al* [15] presented a limit study for module-level parallelism in object-oriented programs. In contrast, in this study, our aim is to illustrate the realizable performance of TLS using a state-of-the-art TLS compiler, while taking into account various TLS overheads.

Kejariwal *et. al* [16] separated the speedup achievable through traditional thread-level parallelism from that of TLS using the SPEC2000 benchmarks assuming an *oracle* TLS mechanism. They [12] later extended their study to the SPEC 2006 benchmarks. It is worth pointing out that they concentrated on only inner-most loops and used probabilistic analysis to predict TLS performance. We also separate the speedup achievable through traditional non-speculative compilation techniques from that requires TLS support; however, we consider all loop-levels instead of just the inner-most or the outer-most loops. Furthermore, they manually intervened to force the compiler to parallelize loops that were not automatically parallelized due to ambiguous dependences. In this paper, we utilize an automatic parallelizing compiler that performs trade-off analysis using profiling information to identify parallel threads—no programmer intervention needed.

While a significant body of previous work focused on exploiting parallelism using TLS at a single loop nest level, relatively little has been done to exploit parallelism at multiple nested loop levels simultaneously. Renau *et. al* [17] proposed hardware-based techniques to determine how to allocate cores to threads that are extracted from different nesting levels; while our paper proposes compiler techniques that statically determines how to schedule loop iterations from different loop nesting levels to different cores. Since the compiler has knowledge of global information, it is able to make better decisions.

III. Dependence analysis of SPEC 2006 loops

Consider the example loop shown in Figure 2(a) with two cross-iteration dependences: a register-based dependence through register $r2$ and a potential memory-based dependence through pointer p and q. In each iteration of the loop, the value of $r2$ from the previous iteration is required, thus the compiler must insert synchronization operations (the `wait/signal` pair) to ensure correct execution (shown in Figure 2(b)). In the case of the memory-based dependence, the cross-iteration dependence only occurs when the load through pointer p accesses the same memory location as the store through pointer q from a previous iteration. Since the compiler is unable to determine the address pointed to by p and q at compile time, it must insert synchronization operations (the `wait_mem/signal_mem` pair) as shown Figure 2(b). However, such synchronization can potentially serialize execution unnecessarily, as shown in Figure 2(c). With the help of TLS, the compiler can parallelize this loop by ignoring ambiguous data dependences and relying on the underlying hardware to detect and enforce all data dependences to ensure correctness at runtime. Figure 2(d) shows the loop executing in TLS mode: when the store through pointer q in $thread1$ accesses the same memory location as the load through pointer p in $thread3$, the hardware detects the dependence violation and restarts the violating thread. $Thread2$, which does not contain the destination of any inter-thread data dependence, is able to execute in parallel with $thread1$. This parallelism cannot be exploited without the help of TLS. However, if the dependence between $store*q$ and $load*p$ occurs frequently causing speculation to fail often, it can potentially degrade performance. In such cases, it is desirable for the compiler to insert explicit synchronization to avoid mis-speculation.

Understanding the inter-thread data dependence patterns in an application is critical for estimating its TLS performance potential. In this section, we analyze the dependence information collected through data dependence profiling, and estimate the importance of TLS hardware support in exploiting parallelism in the SPEC 2006 benchmarks.

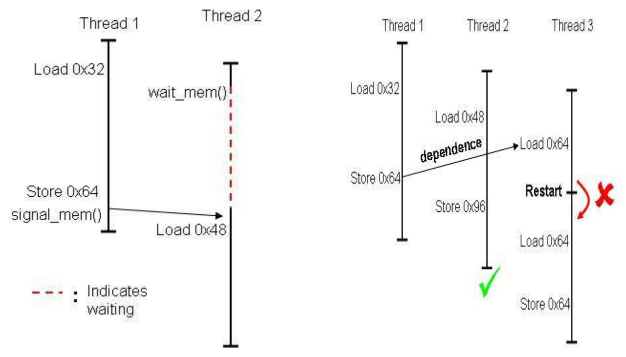
The weight of each loop in an application is summarized as the *combined execution time coverage*, which is defined as the fraction of total execution time of the program spent on a particular loop. In this paper, this weight is estimated using hardware performance counters. To accurately estimate the *combined* coverage of a set of loops, the nesting relationship of these loops must be determined—this is done with the help of a loop tree (for example, Figure 3). An example program and its corresponding loop structure along with profile information is shown in Figure 3. In the example, `loop4`, `loop5` and `loop5'` have no inter-thread memory-based data dependence. The *combined coverage* of loops with no memory-based data dependence is the cumulative coverage of `loop4` and `loop5'`, which is 40%. (Coverage of `loop5` is not included since it is nested inside `loop4`). The loop tree structure used in this paper is similar to the loop graph described by Wang *et. al* [13], except for loops that can be invoked through different calling paths are replicated in loop tree. For example, `loop5` in Figure 3 is replicated, since two different call paths can both lead to the invocation of `loop5`.

```
do  {
   ...
   load *p;
   ...
   r3 = r2 + 2;
   ...
   r2 = r1 + 1;
   ...
   store *q;
} while (condition)
```

(a) A loop with loop-carried register-based and memory-based data dependences.

```
do  {
   ...
   wait_mem()
   load *p;
   ...
   wait()
   r3 = r2 + 2;
   ...
   r2 = r1 + 1;
   signal()
   ...
   store *q;
   signal_mem()
} while (condition)
```

(b) Loop parallelized with synchronization.



(c) Execution serialized due to synchronization. (d) Parallel execution in TLS mode.

Fig. 2. Using synchronization and speculation to satisfy inter-iteration data dependences.

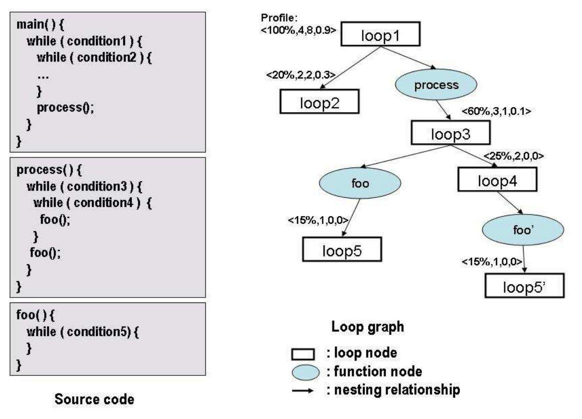
TABLE I
SPEC 2006 benchmarks.

Benchmark	No. of Loops	No. of dynamic loop nesting levels
bzip2	232	11
mcf	52	5
gobmk	1265	22
hmmer	851	5
sjeng	254	10
libquantum	94	4
h264ref	1870	15
astar	116	6
milc	421	11
namd	619	4
povray	1311	15
lbm	23	3
sphinx3	609	8

In this paper, we consider the SPEC CPU 2006 benchmarks written in C or C++ (shown in Table I). We ignore the programs written in FORTRAN since they tend to be parallel scientific programs that can be successfully parallelized using traditional parallelizing compilers and do not require TLS support.

A. *Inter-thread register-based data dependences*

We first focus on the relatively straightforward register-based value dependences. For these dependences, the compiler is responsible for identifying instructions that produce and


```
main( ) {
  while ( condition1 ) {
    while ( condition2 ) {
      ...
    }
    process();
  }
}
```

```
process( ) {
  while ( condition3 ) {
    while ( condition4 ) {
      foo();
    }
    foo();
  }
}
```

```
foo( ) {
  while ( condition5) {
  }
}
```

Fig. 3. An example loop tree showing nesting relationship between loops. Each loop is annotated with four numbers: coverage, number of inter-thread register-based dependences, number of inter-thread memory-based dependences, and the probability of the most probable loop.

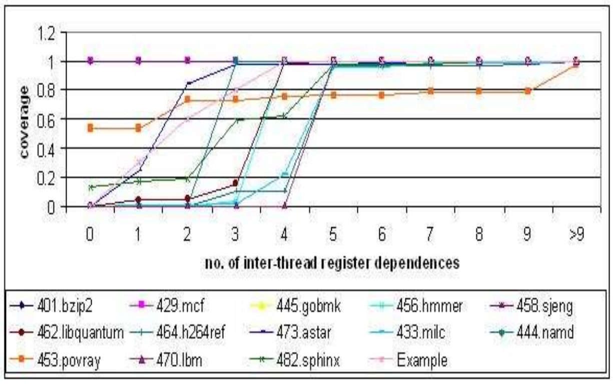
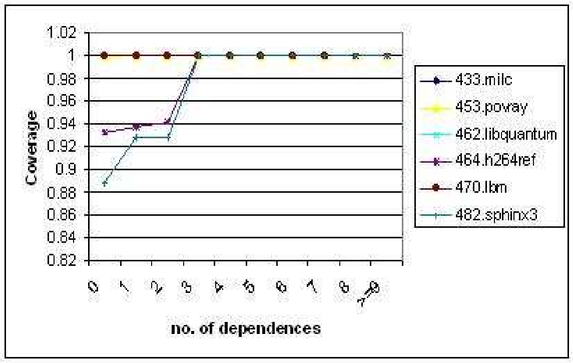
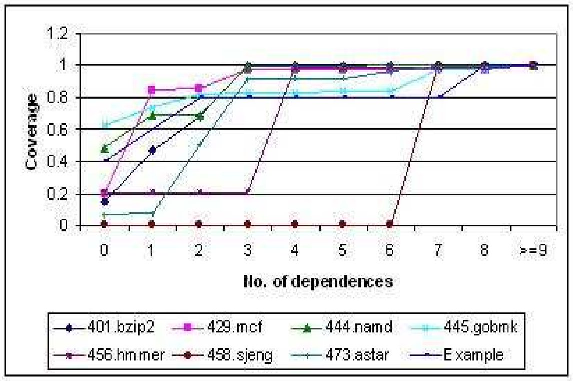


Fig. 4. The combined execution time coverage of loops with inter-thread register-based dependences

consume these value and generate synchronization to ensure correct execution. For example, in the loop shown in Figure 2(a), the compiler identifies the cross-iteration register-based dependence due to register $r2$ and inserts explicit synchronization, as shown in Figure 2(b). We count the number of inter-thread register-based dependences (true dependences) for each loop; and estimate the *combined* coverage of the set of loops with certain number of register-based dependences. The results are presented in Figure 4. The x-axis represents the number of register dependences and the y-axis represents the corresponding *combined* coverage estimated for a certain set of loops. If a benchmark has a combined coverage of C for x number of dependences, it indicates that loops with less than x dependences have a combined coverage of C%. For example, for the loop in Figure 3, the combined coverage of loops with 2 or lesser register dependences is 60%.(coverage of loop2+loop4+loop5). The benchmarks with high combined coverage (C) for a small number of dependences (x), potentially exhibit high degree of parallelism. We found that the high coverage loops in most benchmarks have inter-thread register-based dependences. Thus, an effective TLS compiler that is capable of synchronizing a few inter-thread register dependences is essential. Zhai *et. al* [7] have described how such a compiler can be implemented; and further shown that aggressive compiler scheduling techniques can reduce the critical forwarding path introduced by such synchronizations.



(a) The combined execution time coverage for benchmarks with few inter-thread memory dependences. (Class 'A').



(b) The combined execution time coverage for benchmarks with inter-thread dependences. (Class 'B')

Fig. 5. The combined execution time coverage of loops as a function of the number of inter-thread memory-based data dependences.

B. Inter-thread memory-based data dependences

Unlike register-based dependences, memory-based dependences are difficult to identify using a compiler due to potential aliasing. To ensure correctness, traditional parallelizing compilers insert synchronizations on all possible dependences. With TM or TLS support, the compiler is able to aggressively parallelize loops by speculating on ambiguous data dependences. However, the performance of such execution depends on the likelihood of such data dependences occurring at runtime. If a data dependence does occur, a thread can potentially violate data dependence constraints, and thus must be squashed and re-executed; recovery codes can be executed to restore correct state. For example, there is an ambiguous cross-iteration dependence, shown in Figure 2(a), due to load through pointer $*p$ and store through pointer $*q$. Although the compiler cannot determine whether there is a dependence between $*p$ and $*q$, it can obtain probabilistic information through data dependence profile. In this section, we conduct detailed analysis on inter-thread memory-based dependence using profiling information.

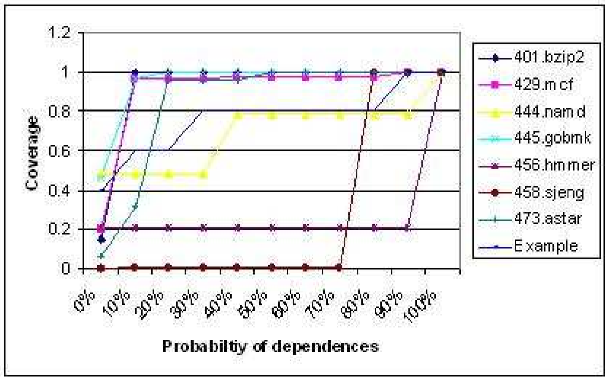
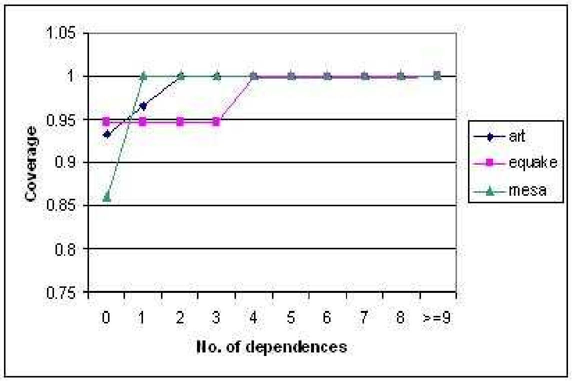


Fig. 6. The coverage of loops with inter-thread memory-based data dependences less than a certain probability.

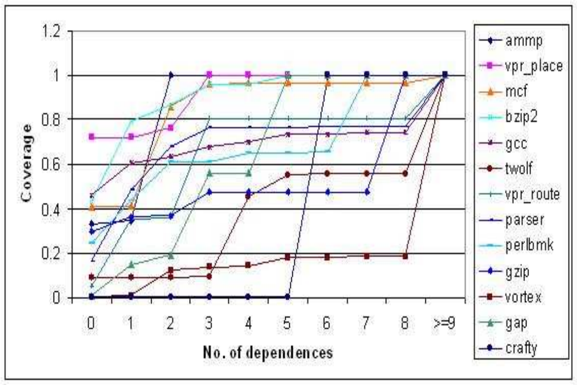
We classify benchmarks based on the combined coverage of loops with different number of memory-based dependences. Figure 5(a) shows the results of benchmarks (points corresponding to 433.MILC, 453.POVRAY, 462.LIBQUANTUM and 470.LBM in Figure 5(a) overlap) that can achieve a high combined coverage with only a few inter-thread memory-based data dependences (class 'A'); Figure 5(b) shows the rest of the benchmarks (class 'B'). For benchmarks in class 'A', 90% or more of the total execution can potentially be parallelized by only considering loops with no inter-thread dependences. These benchmarks can be parallelized without hardware support for speculative execution, if the compiler is able to prove independence between threads.

In class 'B' benchmarks, the speculative hardware support are potentially useful, since inter-thread data dependences do occur. Figure 6 shows the probability of such data dependences and their corresponding coverage for class 'B' benchmarks. The *x*-axis represents the probability of inter-thread memory-based dependences and the *y*-axis represents the corresponding *combined* coverage estimated for a certain set of loops. If a benchmark has a combined coverage of *C* for *x* probability of inter-thread dependence, it indicates the loops that only have inter-thread dependences with probability of less than *x* have a combined coverage of *C*%., For example, for the loop in Figure 3, the combined coverage of loops with only 10% or lesser probability memory dependences is 80%.(coverage of loop2+loop3. Other loops are nested inside loop3). Benchmarks 401.BZIP2, 429.MCF, 445.GOBMK and 473.ASTAR can achieve a large combined coverage, if all loops that only contain data dependences that occur in less than 20% of iterations are speculatively parallelized. These are the loops that could potentially benefit from TLS support.

Some benchmarks, such as 456.HUMMER, 458.SJENG and 444.NAMD, can only achieve a high combined coverage, if loops containing frequently-occurring memory-based dependences are parallelized. These dependences potentially require synchronization. Previous studies has shown that frequently occurring memory-based data dependences could be synchronized by the compiler with profiling data [8]; and aggressive code scheduling could reduce critical-path length introduced by such synchronization [18].



(a) The coverage for benchmarks with fewer inter-thread memory dependences. (Class 'A')



(b) The coverage for benchmarks with significant inter-thread dependences. (Class 'B')

Fig. 7. The coverage of loops with certain number of inter-thread memory-based data dependences in SPEC 2000

C. SPEC 2006 vs. SPEC 2000

In this section, we compare the potential parallelism in SPEC 2000 and SPEC 2006 benchmarks by observing their inter-thread data dependence behavior. Figure 7(a) shows class 'A' benchmarks in SPEC 2000—benchmarks with few inter-thread data dependences; Figure 7(b) shows class 'B' benchmarks—benchmarks with several cross-iteration dependences. Comparing against SPEC 2006 results, shown in Figure 5(a) and in Figure 5(b), we found that SPEC 2000 suite has fewer class 'A' benchmarks. Also the class 'B' benchmarks in SPEC 2000 can only achieve high combined coverage by parallelizing loops with several cross-iteration dependences. Furthermore, by examining Figure 8, which presents the frequency of data dependences that must be speculated during parallel execution, we found that with the exception of AMMP, MCF, VPR_PLACE AND BZIP2, class 'B' benchmarks in SPEC 2000 must speculate on high-probability cross-iteration dependences to achieve a high combined coverage. This is consistent with results reported by previous studies: in SPEC 2000, only a few benchmarks, AMMP, MCF, VPR_PLACE, demonstrated high degree of parallelism under TLS. The data dependences characteristics

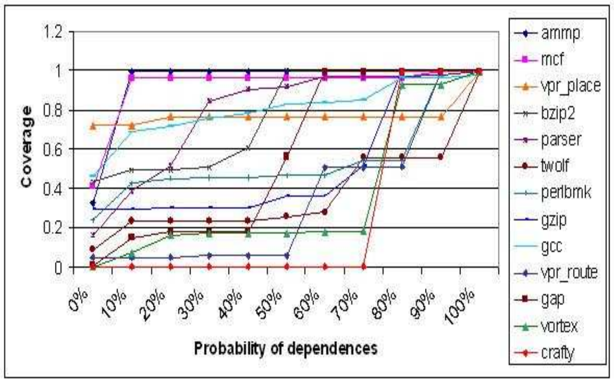


Fig. 8. The coverage of loops with inter-thread memory-based data dependences less than a certain probability in SPEC 2000.

in SPEC2000 and SPEC2006 illustrate that SPEC 2006 can potentially achieve a higher degree of parallelism under the context of TLS.

D. Pitfalls

Even though profiling inter-thread data dependences is crucial in determining the suitability of using TLS to parallelize a loop, TLS performance cannot be directly inferred from this information. In fact, TLS performance depends on many other factors such as the size of the threads, thread spawning overhead, loop iteration counts, and etc. Aggressive code scheduling can reduce the impact of synchronization for inter-thread dependences [7], [8], [13]. Furthermore, library calls can also cause inter-thread data dependences, which is not taken into account here. A common example is the call to *malloc*, which could potentially cause inter-thread dependences due to its internal data structures. Such dependences can potentially be eliminated using parallel libraries.

From the data presented in the earlier sections, we can see that SPEC 2006 benchmarks have numerous inter-thread dependences which could benefit from TLS hardware support. Such TLS hardware support could help to parallelize benchmarks in class ’B’ with low-frequency data dependences and could also help the compiler in handling those ambiguous inter-thread data dependences (in class ’A’ benchmarks). Also, many benchmarks have frequent register and memory dependences which could benefit from aggressive code scheduling by the compiler to reduce critical-path lengths introduced by synchronizations and increase execution overlap between threads.

IV. Compilation and Evaluation Infrastructure

To evaluate the amount of parallelism that can be exploited with hardware support for coarse-grain speculation and advanced compiler optimization technology in the SPEC2006 benchmark suite, we simulate the execution of these benchmarks with an architectural simulator that support multiple cores and speculative execution. In the rest of this section, we will describe the execution model, as well as the compilation and simulation infrastructure used in this paper.

Execution Model:

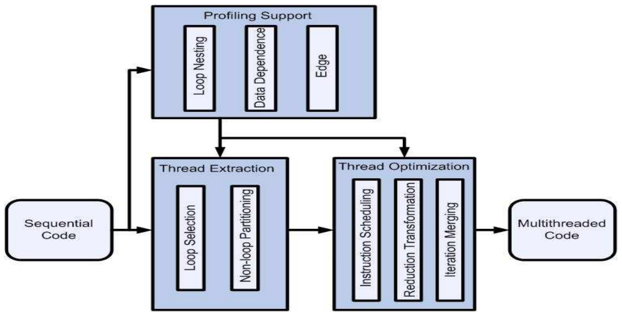


Fig. 9. Compilation infrastructure

Under thread-level speculation (TLS), the compiler partitions a program into speculatively parallel threads without having to decide at compile time whether they are independent. At runtime, the underlying hardware determines whether inter-thread data dependences are preserved, and re-executes any thread for which they are not. The most straightforward way to parallelize a loop is to execute multiple iterations of that loop in parallel. In our baseline execution model, the compiler ensures that two nested loops will not be speculatively parallelized simultaneously. In Section VI, we will study the potential for supporting speculative threads at multiple nesting levels.

We use a cache based protocol based on STAMPede [6] to support speculative threads in CMP. When executing speculative threads, speculative stores will be buffered in the private L1 data cache and speculative loads are marked using a special bit in the cache. When a dependence is detected the violating thread and all its successors are restarted. When the violating thread is squashed, all speculation marker bits in the L1 data cache are reset with a gang-clear (1 cycle). When a thread commits it sends a signal to its immediate successor, and the latter becomes the new non-speculative thread. More details on the TLS architecture model used can be found in [6]. Frequently occurring memory-based dependences and register-based scalar dependences are synchronized by inserting special instructions as shown in Figure 2(b) similar to [8], [7].

Compilation Infrastructure

Our compiler infrastructure is built on Open64 3.0 Compiler [19], an industrial-strength open-source compiler targeting Intel’s Itanium Processor Family (IPF).

To create and optimize speculative parallel threads, the compiler must perform accurate performance trade-off analysis to determine whether the benefit of speculative parallel execution outweighs the cost of failed speculation and then aggressively optimize loops that benefit from speculation. In our case, the compiler performs such analysis and optimizations based on loop nesting, edge, as well as data dependence profiling (using train input set), as shown in Figure 9. The TLS compiler has two distinct phases, as shown in Figure 9, thread extraction and optimization:

- **Thread Extraction:** The compiler can extract threads from both loops and non-loop regions. In this paper we

TABLE II
ARCHITECTURAL PARAMETERS.

Parameter	
Fetch/Issue/Retire width	6/4/4
Integer units	6 units / 1 cycle latency
Floating point units	4 units / 12 cycle latency
Memory ports	2Read, 1Write ports
Register Update Unit (ROB,issue queue)	128 entries
LSQ size	64 entries
L1I Cache	64K, 4 way 32B
L1D Cache	64K, 4 way 32B
Cache Latency	L1 1 cycle, L2 18 cycles
Memory latency	150 cycles for 1st chunk, 18 cycles subsequent chunks
Unified L2	2MB, 8 way associative, 64B blocksize
Physical registers/thread	128 Integer and 128 Floating point registers
Thread overhead	5 cycles for fork/commit and 1 cycle for inter-thread communication
No. of cores	4

focus on loop-level parallelism. In the loop selection phase, the compiler first estimates the parallel performance of each loop, then choose to parallelize a set of loops that maximize the overall program performance based on such estimation. Previous work [20] builds the performance estimation based on detailed data dependence profiling information, as shown in Figure 9. Thus, the achievable performance of speculative parallel threads is tied with the accuracy of performance estimation. I.e., inaccurate profiling information, and inaccurate-estimation information can potentially lead to the selection of sub-optimal loops. In this paper, since we aim to demonstrate the optimal performance that can be achieved with SPEC2006 benchmarks, we eliminated this uncertainty from our evaluation. When selecting which loops to parallelize to maximize program performance, instead of relying on a compiler estimation, we use the simulated parallel and sequential execution time of each loop to determine the actual benefit of parallelizing that loop.

Optimization: Loops that are selected for parallelization must be transformed for efficient speculative parallel execution. In our case, the following optimizations are applied: (i) all register-resident values, as well as memory-resident values that cause inter-thread data dependences in 20% of all threads are synchronized [8]; (ii) instructions are scheduled to reduce the critical forwarding path introduced by the synchronization [7], [13]; (iii) computation and usage of reduction and reduction-like variables are transformed to avoid speculation failure and reduce synchronization [21]; and (iv) consecutive loop iterations are merged to balance the workload between neighboring threads [21].

Simulation Infrastructure:

We simulate a 4-core CMP using a trace-driven, cycle-accurate simulator, where each core is an out-of-order superscalar processor based on SimpleScalar [22]. The trace-generation portion of this infrastructure is based on the PIN instrumentation tool [23], and the architectural simulation portion is built on SimpleScalar. We not only model the register renaming, the reorder buffer, branch prediction, instruction fetching, branching penalties and the memory hierarchy performance, but also extend the infrastructure to model different aspects of TLS execution including explicit synchronization through signal/wait, cost of thread commit/squash, etc. Simulation parameters used for our experiments are shown in Table II.

TABLE III
COVERAGE OF LOOPS PARALLELIZED.

Benchmark	Coverage (%)			No. of loops		
	I	I + II	I + II + III	1	I + II	I + II + III
milc	13	79	79	5	22	22
lbm	0	100	100	0	1	2
h264ref	0	53	83	2	32	36
libquantum	0	98	98	1	5	5
sphinx3	40	83	91	11	19	21
povray	0	3	63	0	4	5
bzip2	2	3	31	4	6	14
mcf	0	85	93	0	6	6
namd	1	8	96	7	22	50
gobmk	0	6	13	0	1	5
hmmer	0	0	79	2	1	6
sjeng	0	0	1	0	0	6
astar	0	5	99	0	2	8

In this study, we use the reference input to simulate all benchmarks. In case of benchmarks with multiple input sets, the first input set is used. To get an accurate estimate of TLS performance, we parallelize and simulate all loops (with at least 0.05% dynamic execution time coverage) in each of the benchmarks. Based on the simulated speedup of each loop, we use our loop selection algorithm to select the best set of loops which maximizes the performance of the entire benchmark. To report the speedup achieved by the entire benchmark, the average speedup of all the selected loops is calculated and weighted by the coverage[1] of the loops. For each simulation run, several billion instructions are fast-forwarded to reach the loops and different samples of 500 million instructions are simulated to cover all the loops.

V. EXPLOITING PARALLELISM IN SPEC2006

In this section, we evaluate the amount of parallelism available in SPEC 2006 benchmarks using the framework described in Section IV. To isolate the parallelism that cannot be exploited without the help of TLS, we take three increasingly aggressive attempts to parallelize loops in SPEC 2006 benchmarks:

Type I: Loops that are identified as parallel by a traditional compiler;

Type II: Loops that have no inter-thread data dependence at runtime (for the particular *ref* input set used), but are not identified as parallel by the compiler, a.k.a., *Probably Parallel Loops*;

Type III: Loops that contain inter-thread data dependences, thus require TLS support to parallelize, a.k.a., *True Speculative Loops*.

[1]The coverage of a loop is defined as the fraction of dynamic execution time of the loop

Table III shows the percentage of total execution that can be parallelized when loops of different types become parallelizable.

To determine the performance impact associated with parallelizing a particular type of loops, the set of loops belong to that type are selected to maximize overall performance. The overall program speedup is then calculated by considering the speedup and coverage of the selected loops. For example, let the selected set of loops be $\{L_1, L_2, L_3, ... L_n\}$. Let their corresponding coverage be $\{ C_1, C_2, C_3, ... C_n \}$ and their corresponding speedup be $\{ S_1, S_2, S_3, ... S_n \}$. The overall program speedup is then calculated as $Speedup = 1/((1\text{-}(C_1 + C_2 + .. C_n)) + C_1/S_1 + C_1/S_2 + .. C_1/S_n)$. In this experiment, we assume it is always possible to identify the optimal set of loops that maximize overall performance, however, in reality, the compiler can potentially select sub-optimal loops due to performance estimation error [18].

A. *Type I Loops*

We applied the Intel C++ compiler [24] to the SPEC 2006 benchmarks to select parallel loops. The benchmarks are compiled with `-O3 -ipo -parallel -par -threshold0` options. The option `-par-threshold0` allows the compiler to parallelize loops without taking into consideration thread overhead. The loops selected by the Intel compiler are then parallelized using our TLS compiler and simulated. The speedup achieved by the selected loops over sequential execution is shown as the first set of bars in Figure 10. With the exception of MILC, which achieved a speedup of 11%, and SPHINX3, which achieved a speedup of 7%, none of the benchmarks is able to speedup over sequential application. Overall, the geometric mean of the speedup is only 1%.

This result is anticipated, since the complex control flow and ambiguous data dependence patterns prohibit the traditional compiler from parallelizing large loops. We have found that in most benchmarks the compiler has only chosen to parallelize simple inner loops with known iteration count. It is worth pointing out that, although many class **A** benchmarks, such as MILC and LBM, contain loops with no inter-thread data dependences, the compiler is unable to identify these loops as being parallel.

B. *Type I + II Loops*

With the addition of *Probably Parallel Loops*, class **A** benchmarks achieve significant performance gain, however, class **B** benchmarks remain sequential. The class **A** benchmarks gain 68% speedup due to these *Probably Parallel Loops* while class **B** benchmarks gain only 4%. If the compiler is able to determine that these loops are parallel, we can potentially parallelize these loops without TLS support. Among the class **A** benchmarks, significant portion of the loops in SPHINX3 and H264REF are *Probably Parallel Loops*; and all loops in MILC, LBM and LIBQUANTUM are *Probably Parallel Loops*.

C. *Type I + II + III Loops*

With the addition of *True Speculative Loops*, we find that many class **B** benchmarks are able to achieve speedup. With only these *True Speculative Loops* class **B** benchmarks gain a speedup of 42% giving them an overall speedup of 46%.

To examine TLS performance in detail, Figure 11 shows the execution time breakdown of parallel execution with TLS support (only selected loops) and sequential execution. The **SEQ** bars show the normalized execution time of the sequential execution running on one core. The **CMP** bars show the normalized execution time of the parallel program executing on four cores. Each bar is divided into six segments: *Busy* represents the amount of time spent in executing useful instructions and the delay due to lack of instruction level parallelism inside each thread; *Lack of threads* represents the amount of time wasted due to the lack of parallel threads (probably due to low iteration count in a loop); *Synchronization* represents the amount of time spent in synchronizing frequently occurring memory dependences and register dependences; *Cache misses* represents the amount of time the processor stalled due to cache misses; *Squash* represents the amount of time wasted executing instruction that are eventually thrown away due to failed speculation; *Other* corresponds to everything else. In particular, it includes time wasted due to speculative buffer overflow and load imbalance between consecutive threads.

We first will focus on the class **B** benchmarks. In HMMER, the loop at `fast-algorithms.c:133` is selected for parallelization, however it has many inter-thread dependences that require synchronizations. These synchronizations create a critical forwarding path between the threads and serialize execution. Thus, by performing speculative instruction scheduling to move the producers of these dependences as early as possible in the execution [7], [13], the parallel overlap is significantly increased; and the benchmark achieves a 90% program speedup. Similar behavior is observed in NAMD, where synchronization and instruction scheduling leads to a 164% program speedup.

For ASTAR, the important loop is at `way2_.cpp:100`, which has a few inter-thread dependences. Some of these dependences are frequent, and thus are synchronized; others are infrequent, and thus are speculated on. Without TLS support, these infrequent occurring dependences must be synchronized, and can lead to serialization of the execution. With the help of TLS, this loop achieves a 17% speedup.

POVRAY, although a class **A** benchmark, is able to benefit from speculation. The important loop in `csg.cpp:248` is a *true speculative loop* with a few mispeculations, thus it is non-parallel for a traditional compiler. Unfortunately, the selected loops have small trip counts, and the cores are often idle; thus the benchmark is only able to achieve a moderate program speedup of 9%.

Not all benchmarks are able to benefit from TLS. GOBMK has many loops with low trip counts, thus many execution cycles are wasted as the cores are idling. Loops with large trip counts are not able to achieve the desired speedup for two reasons: first of all, the amount of work in consecutive iterations is often unbalanced; secondly, many iterations have large memory footprints that lead to buffer overflow of the

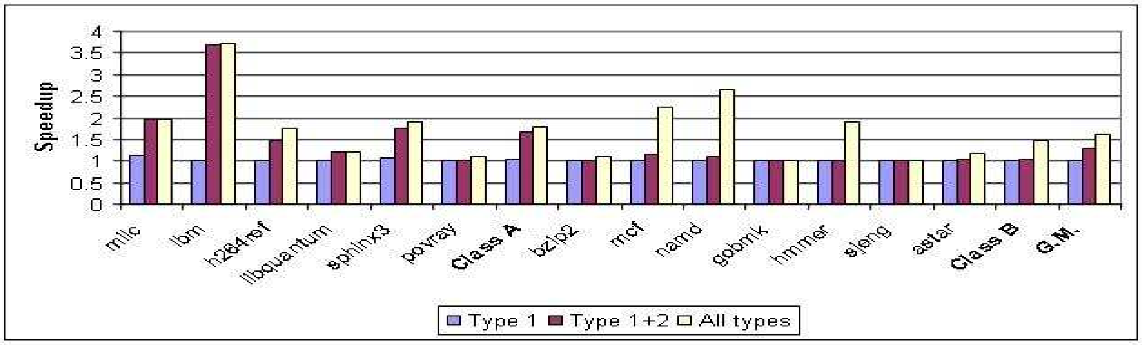


Fig. 10. Shows the program speedup when different types of loops are parallelized using 4 cores.

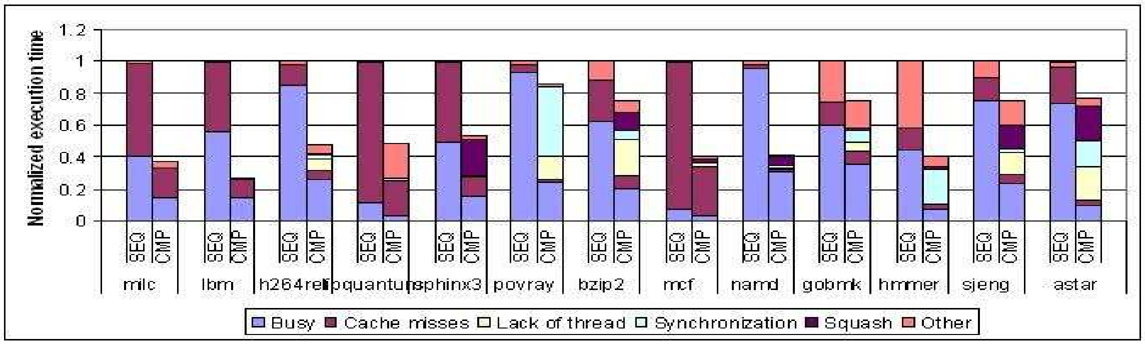


Fig. 11. Shows the breakdown of execution time while executing the selected loops normalized to sequential execution time

speculative states. The geometric mean of the thread size for the top 50 loops (in terms of coverage) is 800,000 instructions. Overall, GOBMK only achieves 1% performance improvement with TLS support.

Loops in SJENG have many inter-thread dependences that occur in 70% of all iterations, and thus need synchronization. However, the critical forwarding path introduced by these synchronization cannot be reduced through instruction scheduling due to intra-thread dependences. Thus, SJENG was unable to benefit from TLS.

To summarize, TLS is effective in parallelizing both class **A** and class **B** loops. Overall, if we select the optimal set of loops, we can achieve a program speedup of about 60% (geometrical mean) , in contrast to a traditional compiler, which only achieves a 1% program speedup.

D. Scalability

As technology scales, the number of cores that can be integrated onto a single die increases. Thus, it is important to understand whether TLS can efficiently utilize all the available cores. In this section, we study the scalability of TLS performance by comparing the speedup achieved using two, four and eight cores. The results of this study are shown in Figure 12.

When the number of cores is increased from two to four, the geometric mean of the speedup increases by about 35%; when increased further to eight cores, the performance increase is 33%. Among class **A** benchmarks, LBM,SPHINX3, H264REF and LIBQUANTUM contain important loops that have large iteration count and substantial amount of parallelism, thus the

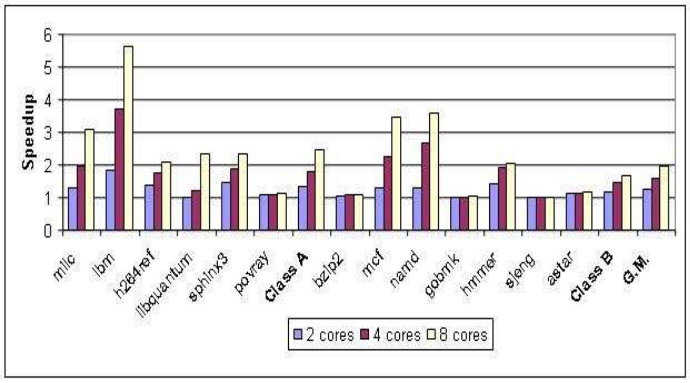


Fig. 12. Speedup increases with increasing number of cores

performance of these benchmakrs is able to scale with the number of cores. In LIBQUANTUM, the super-linear performance gain is due to cache prefetching effect between the speculative threads.

Among class **B** benchmarks, NAMD shows good scalability and MCF benefits from cache prefetching effect as the number of threads increases. Unfortunately, none of the other benchmarks are scalable: HMMER suffers from frequent synchronization; POVRAY and BZIP2 suffers from small trip counts; ASTAR not only suffers from frequently synchronization, but also frequent squashes; For GOBMK and SJENG the performance improvement for TLS is negligible in all configurations.

To summarize, with our existing execution model, only a few benchmarks are able to scale with the number of cores;

and even for the benchmarks that do scale, most of them scale sub-linearly. While the reasons for the lack of scalability differ from benchmark to benchmark, it is obvious that the amount of parallelism is limited. Thus, we will develop new execution models to improve the scalability of TLS in the next section.

VI. Exploiting Speculative Parallelism at Multiple Levels of Loops

As the number of cores on a chip keeps increasing, it is important to ensure that the parallel performance of TLS loops can scale. Unfortunately, almost all benchmarks scale sub-linearly, and many, HUMMER, POVRAY, BZIP2, ASTAR, GOBMK and SJENG exhibits poor scalability. Similar to previous proposals on TLS, the results presented in Section V-D only allow one loop nest level to execute in parallel when multiple levels of nested loops exist. Our goal in this section, is to examine the feasibility and benefit of exploiting TLS parallelism from multiple loop levels in a loop nest simultaneously. Renau *et. al* [17] proposed an architectural design to support out-of-order forking of speculative threads. It allows spawning of more speculative threads on the outer loop levels before spawning less speculative threads on the inner loops. However, it is unclear how to best allocate cores to speculative threads for the outer and the inner loops. In their work, a hardware-based design, referred to as *dynamic task merging* is deployed. Hardware counters are used to identify threads that suffer frequent squashes; and these threads are prohibited from spawning new threads. Other than requiring complex hardware support, the proposed approach has the following limitations. First of all, using number of squashes as a measurement of TLS efficiency is inaccurate. As we have seen in Section V, the performance of TLS loops is also determined by synchronization, iteration count, load balancedness, and etc. Secondly, the hardware is unable to pre-determine the TLS potential of inner loops. As long as the outer loop does not show performance degradation, it will monopolize the cores. The inner loops will never be attempted for parallelization even if they have more parallelism. This will lead to a suboptimal allocation of cores.

In this section, we propose a compiler-based core allocation scheme that statically schedules cores for threads extracted from different levels of a loop nest. In this section, the performance potential of parallelizing multiple levels of TLS loops is extrapolated from the parallel performance of single level of parallel execution using the compiler-based allocation scheme. Since the nested loops are parallelized simultaneously, this extrapolation can be inaccurate, due to some secondary effects such as increase/decrease in cache misses. Furthermore, it is possible for the number of squashes for an outer loop to vary slightly due to the parallelization of inner loops. However, we believe that the impact of these secondary effects can be negligible; and neglecting these effects does not change the conclusion of this paper. Exploring the hardware modifications needed to support the parallelization of multiple levels of loops is beyond the scope of our paper.

Compiler-based scheduling schemes for nested loops have been studied in the past to support nested DOALL and DOACROSS loops. We extend the OPTAL algorithm [25] that was originally designed for core allocation for nested DOALL and DOACROSS loops to allocate cores for TLS loops at compile time.

A. Speculative OPTAL algorithm

The OPTAL algorithm uses a dynamic programming based *bottom-up* approach to decide how many cores to allocate to each loop-nest level so that entire benchmark with multiple-levels of nested parallel loops can achieve the optimal performance. Based on simulated performance and coverage of each loop level, the algorithm considers different possible allocations on TLS loop nests and selects the allocation that maximize program performance. When deciding how many cores to allocate to a TLS loop at a particular loop-nest level, the algorithm only examines its immediate inner loops.

The inputs to the algorithm are the loop tree obtained during loop-nest profiling phase of the compiler and the maximum number of cores available for the benchmark (say 2^K). The output is the optimal core allocation for each TLS loop level and also the estimated optimal performance for the benchmark.

Predicting performance for each loop: Before estimating the multi-level TLS loops performance, the algorithm estimates the single-level performance of each TLS loop level. In the case of a DOACROSS loop, the time required for the parallel execution T_p can be calculated by its initiation delay d. A similar method to estimate the performance of a TLS loop has been studied in [13]. In this study, we use the real performance result for each TLS loop we obtained in the previous section.

Let $SingleSpeedup_{i,j}$ represent the speedup achieved by parallelizing the single-level TLS $loop_i$ using j cores, where $j \in 2^0,2^1,2^2, \ldots 2^K$. Let *BestSpeedup(i,j)* represent the best speedup achievable by parallelizing the multi-level TLS loop nest starting at *($loop_i$)* using j cores.

Combining speedup: The basic step in the algorithm is to find the speedup of an outer loop when its inner loops are also parallelized. Lets call the function to calculate this as *GetCombinedSpeedup(L_i,M,N)*. *GetCombinedSpeedup(L_i,M,N)* returns the speedup of loop L_i when we allocate M cores to parallelize L_i and N cores to parallelize its child loops in the loop tree $L_{i,j}$. *GetCombinedSpeedup(L_i,M,N)* is shown in Algorithm 1.

Recursive algorithm:

The Speculative-OPTAL algorithm starts from the leaf level in the loop tree and calculates the speedup of parent nodes based on the child loops' speedup. The Speculative-OPTAL algorithm is shown in Algorithm 2.

The exact allocation of cores to the inner loop is given in the vector ChildAllocate(L_i,p). This would be used by the compiler to statically allocate cores. Here, we are interested in the value of *BestSpeedup* L_{root},2^K, the performance potential of TLS for the entire benchmark when applied to multiple loop levels.

Complexity analysis:

The Speculative-OPTAL is called for every node in the loop tree. Let λ be the number of nodes in the tree (including both loops and functions). In Speculative-OPTAL, the outer loop

```
Input: Outer loop $L_i$, M cores allocated to $L_i$, N cores
       allocated to the next level ($L_i$'s child loops)
Output: Speedup of $L_i$ with the specified allocation
Read Cycles($L_i$) - the number of cycles spent in loop $L_i$
from the profile;
foreach Child of loop $L_i$, $L_{i,j}$ do
    /*Find total sequential cycles $T_s$ for all inner loops.*/
    Read Cycles($L_{i,j}$) - the number of cycles spent in loop
    $L_{i,j}$ when invoked from $L_i$ from the profile;
    SumBefPar += Cycles($L_{i,j}$);
    /*Find total $T_p$ for all inner loops after
    parallelization.*/
    ParCycles($L_{i,j}$) = Cycles($L_{i,j}$) $\div$ BestSpeedup($L_{i,j}$,N);
    SumAfterPar += ParCycles($L_{i,j}$);
end
/*$T_s$ of outer loop after inner loops are parallelized.*/
CyclesInnerPar = Cycles($L_i$) - SumBefPar + SumAfterPar;
/*Calculate combined speedup.*/
ParCycles($L_i$) = CyclesInnerPar $\div$ $SingleSpeedup_{i,M}$;
return (Cycles($L_i$) $\div$ ParCycles($L_i$)) ;
```

Algorithm 1: The *GetCombinedSpeedup(L_i,M,N)* to get the speedup of outer loop L_i when its inner loops are parallelized.

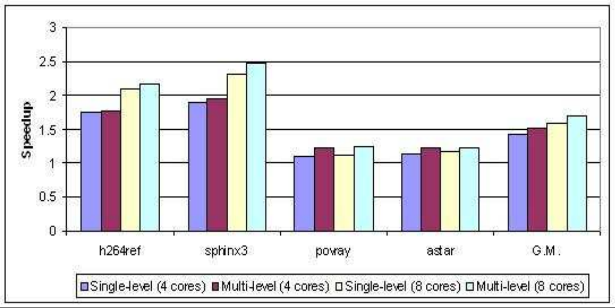


Fig. 13. Speedup over sequential execution using 4 and 8 cores with multi-level TLS loops

iterates for K times and the inner loop iterates on the average of K/2 times. So, total number of times *GetCombinedSpeedup* is called is $K^2/2$. The loop inside GetCombinedSpeedup iterates over all the children of the node. The average number of children per node is a constant (C_1) for a tree. So, the total time taken for Speculative-OPTAL is approximately $\lambda K^2/2$. Therefore, the complexity of Speculative-OPTAL is O(λ) since K is a constant.

B. Results

If TLS threads are spawned only from a single level of loop, many benchmarks, both from class A and class B, exhibit diminishing return as the number of cores increases, as shown in Figure 12. With spawning speculative threads from multiple levels of loops, cores can potentially be better utilized, resulting in better scalability of the cores. We applied our allocation algorithm to extract speculative threads for all benchmarks, but omitted the results for some from our presentation due to the following reasons: BZIP2, GOBMK and SJENG are omitted due to the lack of TLS parallelism overall; LIBQUANTUM and

```
Input: Loop $L_i$ and $2^K$ the number of cores to allocate.
Output: Estimated speedup of the entire benchmark -
        BestSpeedup($L_{root}$,$2^K$) and a vector
        ChildAllocate($L_i$,p) which indicates for each
        loop, how many cores need to be allocated to its
        child loops.
if $L_i$ is leaf then
    foreach p $\in$ { $2^0$, $2^1$, $2^2$, .. $2^K$ } do
        BestSpeedup(i,p) = $SingleSpeedup_{i,p}$;
    end
    return;
end
/*Allocate child loops first.*/
foreach $L_{i,j}$ child of $L_i$ do
    Speculative-OPTAL($L_{i,j}$);
end
/*Inner loop's best allocation already known. Try all
possible allocations for the outer loop*/
foreach p $\in$ { 0,1,..K } do
    foreach q $\in$ { 0,1,..p } do
        CurSpeedup($L_i$,q) =
        GetCombinedSpeedup($L_i$,$2^q$,$2^p/2^q$);
        if CurSpeedup($L_i$,q) $\geq$ MaxSpeedup then
            MaxSpeedup = CurSpeedup($L_i$,q);
            ChildAllocate($L_i$,p) = q;
        end
    end
    BestSpeedup(i,p) = MaxSpeedup;
end
```

Algorithm 2: The *Speculative-OPTAL algorithm.*

HUMMER are omitted due to the lack of nested parallelizable loops; LBM, NAMD and MCF are omitted due to the fact that the Speculative-OPTAL algorithm is unable to identify multiple levels of loops that can perform better than the single level TLS.

For the remaining benchmarks, Figure 13 compares the performance of multi-level TLS performance estimated based on our algorithm with the single-level TLS performance. With four cores, we can extract TLS parallelism from two levels of loops, and with eight cores, we extract TLS parallelism from upto three levels. In SPHINX3 the selected loop cannot utilize all the available cores due to frequent squashes. Multi-level TLS can potentially improve the performance over single-level TLS by about 18% with 8 cores. In POVRAY, the iteration counts of the selected loops are low (around 4), and thus many cores idle. With multi-level TLS, all the cores are utilized, and thus multi-level TLS can potentially outperform single-level TLS by about 13%. In benchmark ASTAR the selected loops have frequent mis-speculations, and thus execution cycles are wasted. With multi-level TLS, two nested loops, in `Way2_.cpp`, `line 65` and `line 100` respectively, are selected, are parallelized simulteneously. The multi-level TLS outperforms single-level TLS by 6% on 8 cores. In H264REF, multi-level TLS outperforms single-level TLS by 7% on 8 cores. Overall, for the benchmarks mentioned, Figure 13 shows that by extracting speculative threads at multiple levels of loop nest, we are able to achieve an additional speedup of about 8% with four cores and 11% with eight cores for these selected benchmarks.

From the above results, it is clear that many SPEC 2006 benchmarks have potential for multi-level TLS. Our compiler-based allocation algorithm can allocate cores among the multi-levels TLS loops to improve overall performance. With such a compiler-based approach, we could also avoid complex hardware modifications needed to support multi-level TLS.

VII. Conclusions

Previous studies of SPEC 2006 based on high level analysis have shown only a limited potential for TLS. These studies did not taken into account the benefits of compiler-based optimizations. In this paper, using a state-of-the-art TLS compiler, we show that SPEC 2006 applications can be successfully parallelized speculatively with TLS.

We show that often the traditional parallelizing compiler cannot prove independence due to the existence of complex control flow and ambiguous data accesses, even if many benchmarks contain parallel loops. With the help of TLS, these *potentially parallel loops* can be parallelized, and thus potentially allowing six benchmarks, MILC, LBM, H264REF, LIBQUANTUM, SPHINX and POVRAY, to achieve a speedup of 78%, if the best set of loops are selected. Furthermore, TLS can parallelize loops that cannot be parallelized by traditional compilers due to infrequent inter-thread dependences (*truly speculative loops*). With TLS, benchmarks BZIP2, MCF, NAMD, GOBMK, HMMER, SJENG and ASTAR can potentially achieve an additional 46% speedup. Overall, with four cores we can achieve a speedup of 60% on all benchmarks (geometric mean) and with eight cores the speedup can reach 91% when compared to sequential execution.

To exploit parallelism at multiple levels of loop nest, we used a novel compiler-based core-allocation scheme to efficiently allocate cores to iterations from multiple levels of a loop nest. Our results show that the proposed mechanism can potentially achieve an additional 11% performance gain with a 8-core processor on selected benchmarks.

Acknowledgements: This work is supported in part by a grant from National Science Foundation under CNS-0834599, EIA-0220021, a contract from Semiconductor Research Cooperation under SRC-2008-TJ-1819, and gift grants from Intel and IBM.

References

[1] J. Emer, "EV8: The Post-ultimate *Alpha*.(Keynote address)," in *International Conference on Parallel Architectures and Compilation Techniques*, 2001. [Online]. Available: http://research.ac.upc.es/pact01/keynote.htm

[2] Intel Corportation, "Intel Pentium 4 Processor with HT Technology," http://www.intel.com/personal/products /pentium4/hyperthreading.htm.

[3] Intel Corporation, "Intel's Dual-Core Processor for Desktop PCs," http://www.intel.com/personal/desktopcomputer/dual_core/, 2005.

[4] AMD Corporation, "Leading the Industry: Multi-core Technology & Dual-Core Processors from AMD," http://multicore.amd.com/en/Technology/, 2005.

[5] J.-Y. Tsai, J. Huang, C. Amlo, D. Lilja, and P.-C. Yew, "The Superthreaded Processor Architecture," *IEEE Transactions on Computers, Special Issue on Multithreaded Architectures*, vol. 48, no. 9, September 1999.

[6] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "The stampede approach to thread-level speculation," in *ACM Trans. on Computer System*, vol. 23, August 2005, pp. 253–300.

[7] A. Zhai, C. B. Colohan, J. Steffan, and T. C. Mowry, "Compiler Optimization of Scalar Value Communication Between Speculative Threads," in *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, Oct 2002.

[8] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry, "Compilerx Optimization of Memory-Resident Value Communication Between Speculative Threads," in *The 2004 International Symposium on Code Generation and Optimization*, Mar 2004.

[9] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai, "A Cost-Driven Compilation Framework for Speculative Parallelization of Sequential Programs," in *ACM SIGPLAN 04 Conference on Programming Language Design and Implementation (PLDI'04)*, June 2004.

[10] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas, "POSH: A TLS Compiler that Exploits Program Structure," in *ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming*, March 2006.

[11] Standard Performance Evaluation Corporation, "The SPEC CPU 2006 Benchmark Suite," http://www.specbench.org.

[12] A. Kejariwal, X. Tian, M. Girkar, W. Li, S. Kozhukhov, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos, "Tight analysis of the performance potential of thread speculation using spec cpu 2006," in *ACM SIGPLAN 2007 Symposium on Principles and Practice of Parallel Programming*, 2007.

[13] S. Wang, "Compiler Techniques for Thread-Level Speculation," Ph.D. dissertation, University of Minnesota, 2007.

[14] J. Oplinger, D. Heine, and M. Lam, "In Search of Speculative Thread-Level Parallelism," in *Proceedings PACT 99*, October 1999.

[15] F. Warg and P. Stenstrm, "Limits on speculative module-level parallelism in imperative and object-oriented programs on cmp platforms," in *International Conference on Parallel Architectures and Compilation Techniques (PACT 2001)*.

[16] A. Kejariwal, X. Tian, W. Li, M. Girkar, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos, "On the performance potential of different types of speculative thread-level parallelism," in *20th Annual ACM International Conference on Supercomputing*, 2006.

[17] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas, "Tasking with Out-of-Order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation," in *19th Annual ACM International Conference on Supercomputing*, June 2005.

[18] S. Wang, A. Zhai, and P.-C. Yew, "Exploiting Speculative Thread-Level Parallelism in Data Compression Applications," in *The 19th International Workshop on Languages and Compilers for Parallel Computing*, Oct 2006.

[19] "Open64 the open research compiler," http://www.open64.net/.

[20] S. Wang, K. S. Yellajyosula, A. Zhai, and P.-C. Yew, "Loop Selection for Thread-Level Speculation," in *The 18th International Workshop on Languages and Compilers for Parallel Computing*, Oct 2005.

[21] A. Zhai, S. Wang, P.-C. Yew, and G. He, "Compiler optimization for parallelizing general-purpose applications under thread-level speculation," in *Poster presented at ACM SIGPLAN 2008 Symposium on Principles and Practice of Parallel Programming*, Feb 2008.

[22] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *ACM SIGARCH Computer Architecture News*, June 1997.

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN 05 Conference on Programming Language Design and Implementation (PLDI'05)*, June 2005.

[24] "Intel c++ compiler," http://www.intel.com/cd/software/products/asmo-na/eng/277618.htm.

[25] C. D. Polychronopoulos, D. J. Kuck, and D. A. Padua, "Utilizing multidimensional loop parallelism on large-scale parallel processor systems," *IEEE Trans. Computers*, vol. 38, no. 9, 1989.