Toward a More Accurate Understanding of the Limits of the TLS Execution Paradigm*

Nikolas Ioannou†, Jeremy Singer‡, Salman Khan†, Polychronis Xekalakis§ ∓, Paraskevas Yiapanis‡
Adam Pocock‡, Gavin Brown‡, Mikel Luján‡, Ian Watson‡, and Marcelo Cintra†

† School of Informatics
University of Edinburgh

‡ School of Computer Science
University of Manchester

§ Intel Barcelona Research Center
Intel Labs Barcelona

ABSTRACT

Thread-Level Speculation (TLS) facilitates the extraction of parallel threads from sequential applications. Most prior work has focused on developing the compiler and architecture for this execution paradigm. Such studies often narrowly concentrated on a specific design point. On the other hand, other studies have attempted to assess how well TLS performs if some architectural/compiler constraint is relaxed. Unfortunately, such previous studies have failed to truly assess TLS performance potential, because they have been bound to some specific TLS architecture and have ignored one or another important TLS design choice, such as support for out-of-order task spawn or support for intermediate checkpointing.

In this paper we attempt to remedy some of the shortcomings of previous TLS limit studies. To this end a characterization approach is pursued that is, as much as possible, independent of specific architecture configurations. High-level TLS architectural support is explored in one common framework. In this way, a more accurate upper-bound on the performance potential of the TLS *execution paradigm* is obtained (as opposed to some particular architecture design point) and, moreover, relative performance gains can be related to specific high-level architectural support. Finally, in the spirit of performing a comprehensive study, applications from a variety of domains and programming styles are evaluated.

Experimental results suggest that TLS performance varies significantly depending on the features provided by the architecture. Additionally, the performance of these systems is not only hindered by *data dependences*, but also by *load imbalance* and limited *coverage*.

1 INTRODUCTION

As device scaling continues to track Moore's law and with the end of corresponding performance improvements in out-of-order processors, *multicore* systems have become the norm. Unfortunately, parallel programming is hard and error-prone, sequential programming is still prevalent, and compilers only auto-parallelize the most regular programs.

Thread-level speculation (TLS) [16, 21, 24, 36, 39] has long been proposed as a possible solution to this problem. In TLS systems the compiler/programmer is free to generate threads without having to consider all possible cross-thread data dependences. Parallel execution of threads then proceeds speculatively and the system guarantees the original sequential semantics of the program by transparently detecting data dependence violations, squashing offending threads, and returning the system to an earlier non-speculative correct state.

Research in the field has progressed in three major directions: architectural extensions (e.g., [8, 9, 10, 32, 34, 37, 38]), compiler support (e.g., [11, 12, 22, 33]), and program behavior characterization [18, 19, 25, 28, 29, 42]. Despite the significant forward progress in the field, the reception of TLS by industrial users is still lukewarm [1, 6].

While all the architectural extension proposals (e.g., [8, 9, 10, 32, 34, 37, 38]) have evaluated the benefits of possible extensions to the baseline TLS architecture, they fail to provide broader and more general insights. On the other hand, while works in program characterization [18, 19, 25, 28, 29, 42] have certainly provided some insight into the potential performance gains of TLS, they still fall short of providing an accurate evaluation of the potential gains of the TLS *execution paradigm*. The reasons for these shortcomings of previous work are threefold. (1) Proposals for new extensions naturally focused on the particular extension proposed and did not investigate how it interacts with other TLS features. Similarly, program behavior studies have focused on only a subset of TLS features and have not investigated how these interact. (2) Quantitative evaluations were often tied to a particular TLS architecture config-

*This work was supported in part by EPSRC under grants EP/G000697/1 and EP/G000662/1 and by the EC under grant HiPEAC-2 IST-217068.

∓ Work performed while author was with the University of Edinburgh.

978-1-4244-9295-4/10/$26.00 ©2010 IEEE

uration and choice of parameters. (3) Benchmark choice was often limited to one particular domain or programming style.

In this paper we address these shortcomings and attempt to provide a better understanding of TLS performance potential. In particular, we provide a study that is, as much as possible, independent of any specific architecture configuration or choice of parameters. Instead, a variety of high-level architectural extensions is explored. In this way individual and combined features can be linked with their respective contribution to the potential performance improvement. Moreover, this study is based on a wide variety of benchmarks coming from the engineering, multimedia, and scientific domains and are written in C, Fortran, and Java.

To summarize, the contributions of this paper over previous TLS studies are as follows:

- We perform an in-depth *implementation-independent* study of TLS performance potential, which more accurately reflects the potential of the *execution paradigm*.
- We evaluate how individual *high-level TLS architectural features* contribute to overall performance gains, both in isolation and combination.
- We evaluate benchmarks from a *variety of application domains and programming styles*.

Experimental results suggest that TLS performance varies significantly depending on the features provided by the architecture. As expected, the benefit of certain features depends on the application domain and the programming style. The benefit also depends on the presence, or not, of other features. Finally, our results confirm that *data dependences* are not the only factor constraining performance. *Load imbalance* and limited *coverage* are also important factors in realizing the full performance potential of TLS.

The rest of this paper is organized as follows. Section 2 provides a brief description of TLS. Section 3 describes the methodology used in our study. Section 4 presents results. Finally, Section 5 discusses related work and Section 6 concludes the paper.

2 BACKGROUND

2.1 Basic TLS Execution

Under the *thread-level speculation* (also called *speculative parallelization* or *speculative multithreading*) approach, sequential sections of code are speculatively executed in parallel hoping not to violate any sequential semantics [16, 21, 24, 36, 39]. Sequential control flow imposes a total order on the threads. The terms *predecessor* and *successor* are used to relate threads in this total order. Stores in speculative threads generate unsafe *versions* of variables that are stored in a *speculative buffer*. As execution proceeds, the system tracks memory references to identify any cross-thread data dependence violation. Any value read from a predecessor thread is called an *exposed read*, and must be tracked since it may expose a read-after-write (RAW) dependence. If a dependence violation is found, the offending thread must be *squashed*, along with its successors. When the execution of a non-speculative thread completes it *commits* and the values it generated can be moved to safe storage. At this point its immediate successor acquires non-speculative status and is allowed to commit. When a speculative thread completes it must wait for all predecessors to commit before it can commit.

Speculative threads are usually extracted from either loop iterations or method continuations. The compiler marks these structures with a *spawn instruction*, so that the execution of such an instruction leads to a new speculative thread. The *parent* thread continues execution as normal, while the *child* thread is mapped to any available core. For loops, spawn points are placed at the beginning of the loop body, so that each iteration of the loop spawns the next iteration as a speculative thread. For method calls, spawn points are placed just before the method call, so that the non-speculative parent thread proceeds to the body of the method and a speculative child thread is created from the method's continuation.

2.2 Architectural Extensions

Below we outline the most important, previously proposed, architectural extensions to the basic TLS execution. In Section 4 we evaluate the potential performance benefits of all of these extensions quantitatively.

2.2.1 Multiversioned Cache

A *Speculative Versioning Cache* (SVC) [15] can hold state from multiple tasks by tagging each cache line with a version ID, which identifies the task to which the line belongs to. Multiversioned caches are beneficial in two ways: they avoid processor stalls when tasks are imbalanced, and enable lazy commit. If tasks have load imbalance, a processor may finish a task while it is still speculative. If the cache can only hold state for a single task, the processor has to stall until the task becomes safe. An alternative is to move the task state to some other buffer, but this leads to a more complicated design than a multiversioned cache. Lazy commit is an approach where, when a task commits, it does not eagerly merge its cache state with main memory. Instead, the task simply passes the commit token to its successor. Its state remains in the cache and is lazily merged with main memory later, usually as a result of cache line replacements.

2.2.2 Out-of-Order Spawn

In early TLS proposals, threads are formed in-order from iterations from a single loop level or the continuation of subroutine calls. This may significantly limit the amount of TLP that can be exploited. Recently, architectural support for task creation from nested loops and nested methods has been proposed [34]. This optimization is called *out-of-order* (OoO) *spawning*, since the threads are spawned in a different order than their sequential semantics. Thread ordering for these systems is typically maintained via splitting timestamps [34].

2.2.3 Dynamic Dependence Synchronization

Dynamic dependence synchronization [27] is an important optimization for TLS systems, since it removes much of the overhead associated with recurring dependence violations. The idea is to dynamically detect which instructions violate the sequential semantics and force them to wait until the datum can be forwarded to them. Unfortunately, one cannot indiscriminately synchronize on all memory operations, since each time this is erroneously done, the consumer part of the dependence chain is unnecessarily stalled. We thus have to correctly identify the dependence chains. Most predictors studied so far in the literature are simple, table-based predictors indexed by either the Program Counter of the offending instructions [27] or their target memory addresses [9, 38].

2.2.4 Intermediate Checkpointing

Intermediate checkpointing schemes aim to reduce the misspeculation penalty by allowing partial rollback. They do so by trying to detect the violating loads and checkpoint the processor state *just before* these loads are executed. These schemes are thus guided from a dependence predictor [41], or some other heuristic (e.g., periodically, after a certain number of instructions have executed since the last checkpoint [10]). Once a positive prediction is obtained, the processor state is checkpointed and a subthread is created which has as its first instruction the predicted load. It is important to note that ignoring overheads and prefetching, perfect synchronization (Section 2.2.3) and perfectly placed checkpoints have the same effect.

2.2.5 Data Value Prediction

Normally in TLS systems values have to be forwarded from less speculative to more speculative threads. This can happen either by means of implicit forwarding (possibly following a squash and restart) or explicit synchronization (Section 2.2.3). There are, however, cases where due to value locality this is not necessary, since the values to be communicated can be easily predicted [9, 38]. In such cases squashes can also be avoided if the predicted value of the datum prematurely read is the same as the one that is subsequently stored, and as such the squash dictated by the TLS protocol is unnecessary.

2.2.6 Return Value Prediction

Another form of value prediction that is more specific to method-level speculation is return value prediction [7]. Under this optimization, the return value of a method is predicted. Although this optimization can be seen as a specialization of value prediction, it has been shown to be a fairly important performance optimization on its own.

2.3 Compilation Extensions

Next we outline the most important, previously proposed, extensions to the basic TLS compilation schemes. In Section 4 we evaluate a subset of these extensions.

2.3.1 Extracting Threads Beyond Program Structures

While most previous work on TLS has considered only threads extracted from high-level programming constructs, such as loops and methods, some works have considered other sources. For instance, the work in [17] considers all basic block boundaries as possible thread spawn points and uses a min-cut algorithm based on expected dependences to find the best selection of threads. Similarly, the work in [26] considers all control quasi-independent points (i.e., points more or less guaranteed to postdominate a certain program point) in the program as possible thread spawn targets. Such thread extraction approaches add flexibility to the traditional loop and method approaches, but also add significant complexity to the TLS compilation process. We leave a limit study of such extensions to future work.

2.3.2 Profiling Based Thread Selection

Given the unpredictability of TLS execution, in particular due to data dependence violations, the vast majority of compilation schemes for TLS have used profiling to select the most profitable threads [13, 22, 33]. As is the general case with profiling approaches to compilation, these are affected by run-time variation with different input sets, but have proven very effective in practice. In Section 4 we evaluate the effect of a simplified profiling based selection of loops from different nesting levels.

2.3.3 Thread Selection with Static Cost Analysis

Despite the high unpredictability of TLS execution, some attempts have been made toward static thread selection [12, 20, 40]. Most of these use simple heuristics, such as expected thread sizes, to select threads most likely to be profitable [20, 40], while others have used more complex cost functions [12]. In this work we do not explicitly consider the effects of such static analysis, but our limit study with perfect thread selection from different nesting levels somewhat subsumes part of the gains possible with such techniques.

2.3.4 Pre-Computation Slices

Some previous works have proposed the use of precomputation slices for handling statically known data dependences [33]. The idea is to add code to be executed at the beginning of a thread to generate the live-in values and avoid data

Imperative (C/Fortran)			Object-Oriented (Java)	
SPECINT2006	**SPECFP2006**	**MediabenchII**	**SPECJVM98**	**DaCapo**
400.perlbench	416.gamess	mpeg4enc	_201_compress	antlr
401.bzip2	454.calculix	mpeg4dec	_202_jess	bloat
403.gcc	459.GemsFDTD	mpeg2dec	_205_raytrace	fop
429.mcf	465.tonto	jpg2000enc	_209_db	pmd
458.sjeng	470.lbm	jpg200dec	_213_javac	
462.libquantum	482.sphinx3	cjpeg	_222_mpegaudio	
		djpeg	_228_jack	

Table 1: Benchmarks.

dependence violations. We do not explicitly consider this compilation extension, but our limit study with perfect value prediction somewhat subsumes the gains possible with such techniques.

2.3.5 Compiler Inserted Dependence Synchronization

Some previous works have proposed the use of statically placed dependence synchronization to avoid data dependence violations [38, 43]. These techniques are not as flexible as dynamic approaches to synchronization (Section 2.2.3) but require simpler hardware. We do not explicitly consider this compilation extension, but our limit study with perfect synchronization completely subsumes the gains possible with such technique.

3 METHODOLOGY

In this section we present our simulation methodology for evaluating TLS and the proposed architectural features listed in the previous section. There are conceptually four steps in our methodology: (1) workload selection, (2) annotation of applications to produce (3) sequential traces, and (4) performance evaluation from the traces. The remainder of this section presents each step in detail.

3.1 Benchmarks

Having a representative workload is of paramount importance for any limit study, and as such we feel that this is one of the most important steps in our methodology (and a fundamental limitation in previous ones). An overview of the selected benchmarks can be seen in Table 1. We broadly categorize the chosen benchmarks in imperative applications (C and Fortran) and object-oriented ones (Java).

For the imperative applications, we use benchmarks from the SPEC CPU 2006 [2] benchmark suite running the train data set and from the Mediabench II [14] benchmark suite running the default data set. All benchmarks are compiled with optimization level `O2`.

Our object-oriented applications are Java benchmarks from the SPEC JVM 98 [3] and DaCapo [5] benchmark suites. We use data sets supplied along with the suites, specifically `s1` for SPEC JVM 98, and `small` for DaCapo.

3.2 Instrumentation

The tasks are selected from high-level program structures, consisting of loop iterations and method call continuations. We annotate such program structures at compile time. We chose two distinct compiler infrastructures to perform our experiments, one for the C and Fortran benchmarks and one for the Java ones.

For C and Fortran benchmarks we have used the *GNU Compiler Collection* (GCC) v4.3.3.We have created a compiler pass to annotate loop iterations and method call bodies. Register spilling is done at task boundaries so that all inter-task register dependences are communicated through memory. References to loop induction and reduction variables are also marked in the compiler and dependences carried by these references are not considered, since such dependences can be removed by compiler transformations [30]. The use of return values is also marked. Our annotation operates on the intermediate representation (tree SSA) of GCC at the end of the loop optimization passes, so loops, induction and reduction variables might not directly correspond to the original source code. We chose this option because we think it is more realistic to use optimized code instead of instrumenting at source level and interfering with compiler optimizations.

For Java benchmarks we have modified the Jikes RVM compiler v2.9.3 to instrument the object code. Again, we annotate loop iterations, method call bodies, and references to loop induction and reduction variables. Because Jikes RVM is an adaptive compilation system, we execute each Java benchmark for several warm-up iterations before instrumenting the code on a steady-state iteration, when hot methods have been compiled to a higher optimization level. Finally we eliminate all JVM runtime activity (garbage collection, etc) from the sequential trace files.

3.3 Trace Generation

Having the benchmarks annotated by the compiler, we next create the execution traces. The sequential benchmark traces were produced using Simics [23], a full-system functional simulator. The simulated processor is a single-issue in-order x86 core, where each instruction takes one cycle to execute. The simulator recognizes magic instruction sequences which execute call-backs from the simulated application to the simulator. We employ these to mark dynamic events in program execution, such as method call boundaries. We use the Simics memory profiling infrastructure to trace memory accesses. For C and Fortran benchmarks, we record all memory accesses. On the other hand, for Java benchmarks we only record (global) accesses into the heap, and stack-based accesses occurring directly in loop iterations. Note that in Java, a method's stack frame is guaranteed to be private to that method. Each event in the trace file has an associated sequential time-stamp, derived from the Simics cycle counter, which is adjusted to remove the overhead of each instrumentation code sequence.

3.4 Trace-Driven Simulation

The last step in our methodology is to feed the produced traces into our simulation infrastructure. In order to evaluate the performance impact of different high-level TLS architectural features we have crafted a trace-driven simulation tool. The tool parses the trace file, extracting threads along the way out of loop iterations and/or method call continuations, runs them in parallel with an infinite size *speculative buffer* and dynamically detects data dependences.

3.4.1 TLS Architecture Model

The architectural design options specific to TLS that our framework is able to accurately simulate cover what we believe are the most important TLS-related hardware optimizations. More specifically, using our tool we are able to simulate the effect of: multiversioned caches, out-of-order thread spawning, dynamic dependence synchronization, checkpointing, and data and return value prediction.

For dynamic dependence synchronization, data and return value prediction, the predictors we simulate are perfect. This choice was made because we only want to show an upper bound on performance relative to each feature, as opposed to evaluating different kinds of predictors. Note that, because we do not simulate prefetching effects, perfect synchronization and no overhead checkpointing have the same net effect on performance, such that the results presented for perfect synchronization in Section 4 actually refer to both.

3.4.2 Task Selection

The optimal partitioning of programs into non-speculative threads given the communication costs and the amount of computation associated with each method invocation, has been shown to be NP-Complete [35]. The partitioning of speculative tasks is further complicated because dependence and communication information is not complete. Thus, we rely on heuristics to perform task selection using both loop iterations and method call continuations. In the case of in-order loop-level speculation we consider two alternatives. The first is to evaluate innermost loops. The second is to choose the best loops to speculate on from three different dynamic depth levels, which somewhat emulates the result of an optimal profile or cost analysis driven compiler loop selection. In the case of out-of-order spawning, the dynamic thread spawning policy employed was one favoring the least speculative threads [34]. The spawning policy also takes into account the maximum thread size, and only spawns threads whose length is below a threshold. The threshold was chosen empirically based on the resulting speedup and is different for loop and method-level speculation. For loop-level speculation, we have used an upper bound of 500 instructions, effectively choosing smaller tasks. For method-level speculation we have used an upper bound of one million instructions, with smaller thresholds providing limited coverage.

4 EXPERIMENTAL RESULTS

4.1 Loop-Level Speculation

Loops have traditionally been the center of the research community's effort for parallelization and, hence, are the obvious choice for thread-level speculation as well. We first consider the simpler case of speculatively parallelizing the innermost loops only and then evaluate the potential of choosing between multiple dynamic loop depth levels.

4.1.1 In-Order Loop Speculation

Initially, from each loop nest we only choose the innermost loop to speculate upon. This is the most commonly studied case and is also the starting point of our evaluation. We speculatively parallelize *all* innermost loops in each of the benchmarks.

The results for this type of speculative parallelization are shown in Figure 1. It can be seen that there is significant potential from innermost loops for *lbm*, *libquantum* and *cjpeg*. That is because those benchmarks have high coverage parallel inner loops (99%, 94% and 95%, respectively). The lower scaling of *cjpeg* at 16 processors, however, can be attributed to low iteration counts for some of the innermost loops as well as one loop with 12% coverage that shows infrequent dependences. Detailed high coverage loop information for selected benchmarks is presented in Table 2.

Multiversioning provides minimal to no improvement over base TLS for this type of speculation. This comes as little surprise, since we did not expect innermost loops to be particularly load imbalanced (Section 4.4 provides further discussion on this).

Synchronizing around data dependences benefits *lbm* strongly. The reason is that even though the loop with 90% coverage is parallel, the remaining execution requires synchronization to obtain overlap, most significantly the two loops with the next higher coverage (4% and 2%) [1] require synchronization to be parallelized. Other benchmarks that benefit from synchronization are *mpeg2dec*, *djpeg* and, to a lesser degree, *gamess*. *mpeg2dec* has a 28% coverage loop which shows a speedup of 11 for 16 processors only if synchronized. Similarly, *djpeg* has a 26% coverage loop that shows no parallelism unless synchronized around data dependences. *gamess* has a 22% coverage loop that benefits significantly from synchronization.

Interestingly, value prediction does not seem to provide much benefit on top of synchronization. This suggests that for such fairly regular inner loops the producer and consumer in a data dependence occur very close in time such that synchronization incurs minimal overhead.

Unsurprisingly, Java applications do not show much potential from innermost loops. This is due to low coverage of in-

[1] Since 90% of the program is already parallel, these loops, which might appear to be unimportant, can now make a large contribution to speedup.

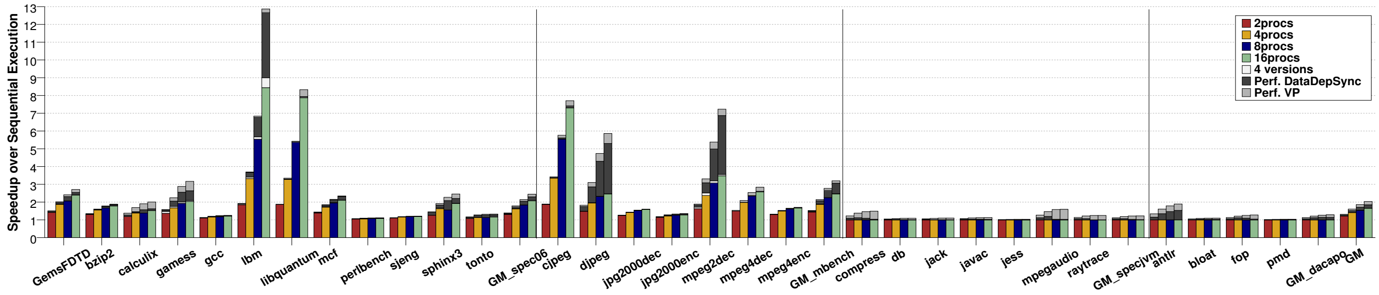

Figure 1: Speedup obtained from In-Order innermost Loop-Level Speculation. Each bar corresponds to a different number of processors. Stacked on top of the bars are the improvements in speedup obtained from 4 versions instead of 1, Perfect Data Dependence Synchronization and Perfect Value Prediction, respectively.

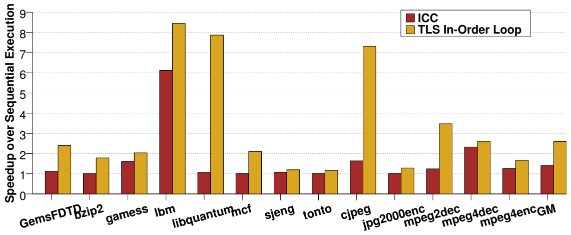

Figure 2: Comparison between parallel loops detected by ICC and TLS In-Order innermost Loop-Level speculation.

nermost loops for the benchmarks evaluated, ranging from 4% (*jess*) to 51% (*antlr*), with an average of 23%.

To verify that the speedup obtained through speculative parallelization is not simply through loops that are parallelizable through current compiler techniques, we compare with Intel's ICC compiler in Figure 2. For a fair comparison, we compute the speedup obtained by the compiler by simulating the same way as earlier, but only parallelizing loops detected as parallel by the compiler. The only benchmarks that the compiler is able to automatically parallelize to any significant degree are *lbm* and *mpeg4dec*.

Beyond innermost loops we also evaluate higher levels of nesting. We do this by picking the best performing level [2] of the first three dynamic loop depths. Figure 3 depicts the results for the best performing loops in a nesting level. *libquantum* continues to get most of its speedup from the high coverage inner loop. However, considering higher nesting levels adds a number of other lower coverage loops which takes the total coverage from 94% to 99%. Similarly, for *mcf* the coverage of the loops contributing is increased from 64% to 94%. In the case of *djpeg*, adding higher level loops increases the coverage and the speedup in the base case. However, these higher level loops suffer from low iteration counts which limit the scalability; this explains why there is no longer any improvement from synchronization or perfect value prediction. For *jpg2000enc* we see a dramatic increase in coverage from 42% to 98% from considering outer loops; these new loops, however, need synchronization and value prediction for realizing their potential. The Java benchmarks continue to be mostly coverage limited and, in some cases (e.g., *mpegaudio*) have high coverage loops with limited speedup due to load imbalance.

4.1.2 Out-of-Order Loop Speculation

In an effort to better exploit loop level parallelism, we also evaluate simultaneously speculating on multiple levels of the same loop nest. This entails spawning out-of-order tasks. The task selection performed is the one presented in Section 3.4. Picking loops to speculate on in this way, we see mixed results, shown in Figure 4. Because we favor safer threads, in loop-level speculation this translates to choosing inner loops most of the time. The only case where this differs is when the inner loops suffer from low iteration count. If the iteration count is low enough, and given enough cores, then the outer loop is speculatively parallelized. In some cases this is beneficial, like *sjeng*, *cjpeg*, *mpeg2dec*, and the *jpg2000* benchmarks. In other cases, if the thread size of the outer loop is significantly different from the inner loop this may result in load imbalance. We observe a slowdown due to this effect compared to choosing the best single level per nest for *libquantum*, *mcf*, and *sphinx3*. The increase in the improvement observed due to multiversioning is also a repercussion of the load imbalance imposed, in some cases, by the task selection policy.

4.2 Method-Level Speculation

4.2.1 In-Order Method Speculation

We evaluate speculatively overlapping method continuations with the methods. We assume perfect return value prediction for methods. As seen in Figure 5, in-order method level speculation is constrained by real dependences that lead to serialization in the presence of synchronization. Data value predic-

[2] When picking the best level we don't take into account synchronization or value prediction.

Program	Loop Location	Coverage	Depth	Avg Iter. Count	Avg Size	TLS Speedup	Perf. Data. Dep. Speedup	Perf. VP. Speedup
lbm	lbm.c:187	90.9	1	1300K	262	12.7	12.7	12.7
	lbm.c:460	4.04	1	1300K	147	1.81	14.4	14.8
	lbm.c:105	2.10	1	100	257	1.36	7.82	14.1
libquantum	gates.c:96	56.3	1	2048	16	15.4	15.4	15.4
	gates.c:65	15.2	1	2048	36	16.0	16.0	16.0
	gates.c:174	11.7	1	2048	17	16.0	16.0	16.0
	gates.c:NA	7.6	2	2048	358	16.0	16.0	16.0
gamess	rhfuhf.fppized.f:410	22.7	1	13K	186	1.91	6.88	15.9
mcf	pbeampp.c:167	40.7	1	300	22	14.3	14.4	14.4
	implicit.c:285	17.0	2	3K	56	2.49	5.50	10.6
	pbeampp.c:81	16.3	2	7	21	3.35	3.43	3.44
cjpeg	jccolor.c:149	33.4	1	704	50	16.0	16.0	16.0
	jcdctmgr.c:185	18.1	1	64	18	14.7	14.7	14.7
	jccoefct.c:144	44.8	3	1.33	4K	1.33	1.33	1.33
	jchuff.c:477	12.7	1	64	23	2.42	8.03	11.7
	jfdctint.c:220	10.9	1	8	88	7.92	7.92	7.92
	jfdctint.c:155	10.5	1	8	84	7.92	7.92	7.92
djpeg	jdcolor.c:145	38.4	1	704	95	16.0	16.0	16.0
	jdsample.c:379	26.2	1	350	108	1.03	7.57	15.9
	jidctint.c:278	15.2	1	8	80	7.91	7.91	7.91
	jdsample.c:378	26.2	2	2	9K	2.00	2.00	2.00
	jdcoefct.c:193	27.6	3	1.33	1.7K	1.33	1.33	1.33
jpg2000enc	jpc_t1ecn.c:871	19.1	2	57	1.8	1.24	2.20	9.57
	jpc_t1enc.c:472	15.6	2	56	2.7K	1.48	1.57	10.3
	jpc_cs.c:941	14.7	3	704	6.5K	1.23	1.53	16.0
	jpc_qmfb.c:595	8.65	2	248	78K	1.37	4.19	16.0
	jpc_dec.c:1091	7.54	2	794	187	1.61	10.6	15.7
mpeg2dec	store.c:259	28.9	1	704	360	1.79	11.4	15.9

Table 2: High coverage loop information for selected benchmarks. The speedup numbers presented are for 16 processors and the loops for each benchmark are ordered based on their Perfect Value Prediction speedup contribution.

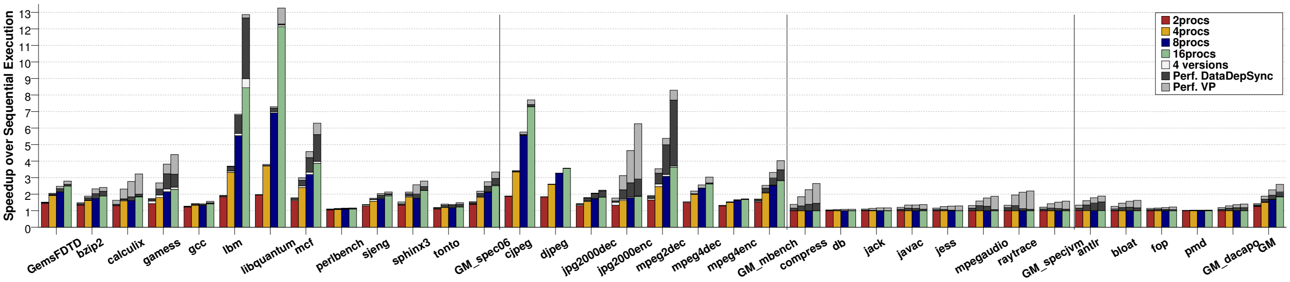

Figure 3: Speedup obtained from choosing the best loop for each loop nest out of three loop depth levels.

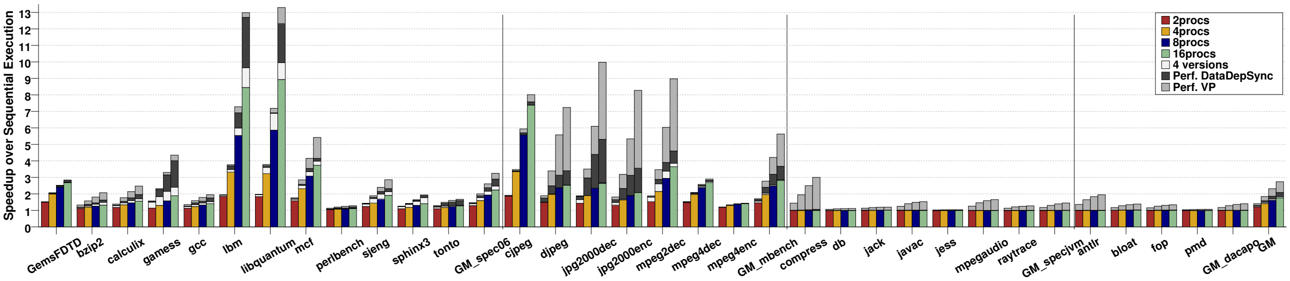

Figure 4: Speedup obtained from Out-of-Order Loop-Level Speculation.

tion has to be employed if significant performance is to be obtained, with the *specjvm98* benchmarks showing the most potential. *mpegaudio* is an exception to this, and shows reasonable speedup with base TLS that can be significantly improved with multiversioning.

4.2.2 Out-of-Order Method Speculation

Figure 6 shows out-of-order method level speculation. This means that nested method calls can be speculated upon as well. This leads to an improvement in speedup with base TLS for a number of benchmarks compared to in-order method-level speculation. Multiversioning is essential for realizing this. However, we no longer see significant improvement from value prediction for most of the benchmarks, suggesting that our OoO task selection leads to increased load imbalance and/or lower coverage. Looking ahead to Figure 9 in Section 4.4, we see that OoO method-level speculation exhibits lower coverage than in-order method-level speculation (e.g., *gcc*, *libquantum*, *jess*, *pmd*, etc). The two exceptions to the decrease in value prediction potential are *djpeg* and *cjpeg*, which exhibit both better base TLS speedup *and* increased potential from value prediction. Moreover, synchronization is also very beneficial in those two cases.

The difference in behavior between in-order and OoO method level speculation is to be attributed to the task selection that each of them entails. In the in-order case, when a method is chosen to be speculatively parallelized, only its continuation (being the most speculative task) is allowed to further spawn more speculative tasks and this is recursively repeated until we fill the processors contexts, assuming enough tasks. On the other hand, OoO coupled with our greedy task selection policy that favors safer threads, will mostly spawn speculative tasks *within* the chosen method(s), assuming enough method calls within the speculated method(s).

4.3 Mixed Speculation

Speculating at both loop iterations and method call continuations is the next form of speculation we consider. The spawning policy we employ in this case is a run-time out-of-order one similar to the one used in loops and methods separately. The difference now is that we can spawn threads from both types of program structures at the same time. We retain the static thread size heuristic that showed most potential, i.e., the one used for loops for SPEC and Mediabench and the one used for methods for the Java benchmarks.

As can be derived from Figures 7 and 8, applying a joint speculation mode can combine the benefits of loop and method level speculation. For most of the benchmarks we simply retain the benefit obtained from either loop-level or method-level speculation. In some cases we see combined performance improvements (*gcc*, *mpeg2dec*, and *mpeg4*). However, we see negative effects from joint speculation for the *mpegaudio* and, to a lesser extent, *jpg2000* benchmarks. If we compare the results obtained with in-order against OoO for mixed speculation we can see that OoO spawning, with the greedy task selection policy employed, does not provide notable benefits over in-order for most of the benchmarks.

4.4 Limits on Performance

The overriding pattern to be seen in the results shown in the previous sections is that performance is not constrained primarily through *data dependences*. Although some of the benchmarks show significant improvement through perfect value prediction, many of them do not. This fact points toward the conclusion that we are also limited by *low coverage* (i.e., large portions of the code are not chosen for speculative execution either because of the granularity of the tasks or because of a task selection heuristic) and *load imbalance* (i.e., co-scheduled speculative tasks have diverse sizes).

As a first step to understanding the performance limitations that handicap speculative execution we try to quantify the effect of *load imbalance*. We compute *load imbalance* as the relative difference between the theoretical maximum speedup obtained using Amdahl's law [4] and the speedup obtained with perfect data value prediction. The intuition behind this is that in the presence of perfect value prediction the only limiting factor to achieving the theoretical speedup is *load imbalance*. The coverage used in the calculation of Amdahl's law is the combined coverage of speculated loops for in-order loop speculation and

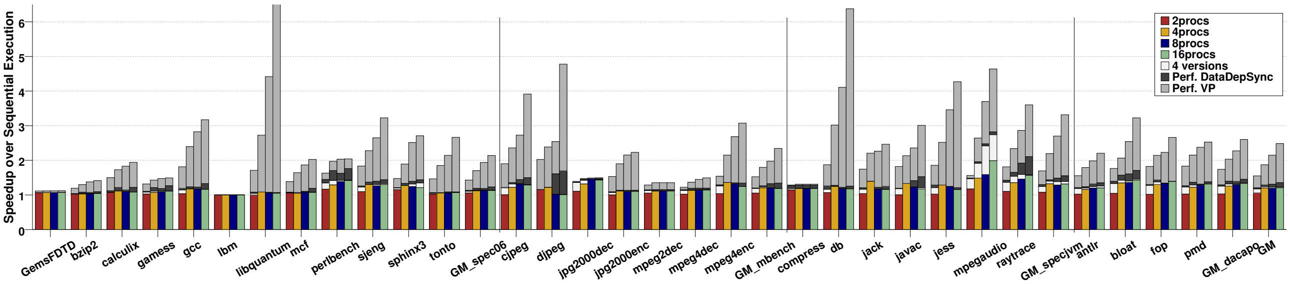

Figure 5: Speedup obtained from In-Order Method-Level Speculation.

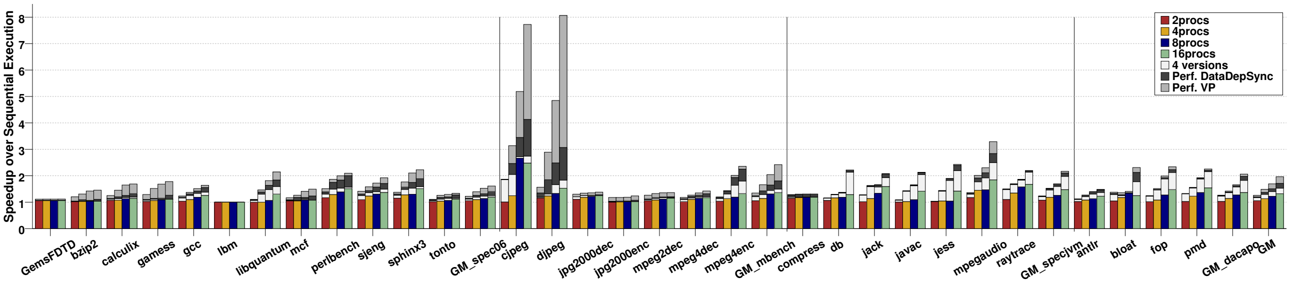

Figure 6: Speedup obtained from Out-Of-Order Method-Level Speculation.

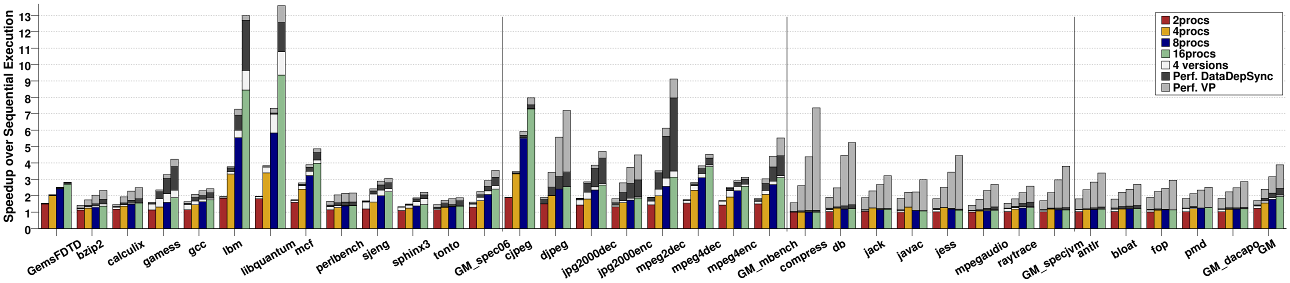

Figure 7: Speedup obtained from In-Order Mixed Speculation.

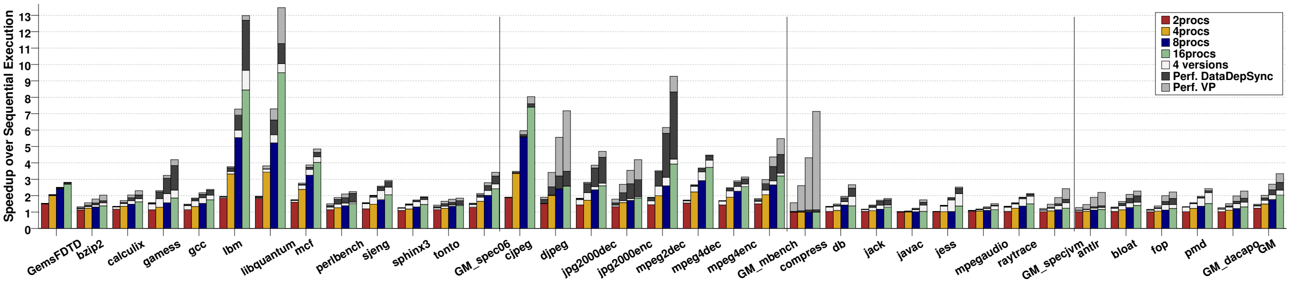

Figure 8: Speedup obtained from Out-Of-Order Mixed Speculation.

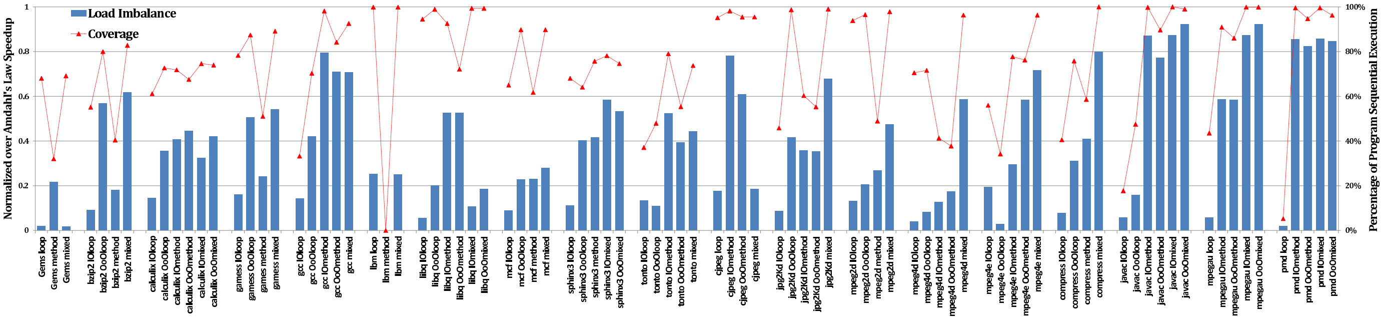

Figure 9: Relative Load Imbalance (bars) and Coverage (line), for 16 processors. For benchmarks showing nearly identical behavior we show only a representative one. Identical OoO and In-Order results are also merged.

the combined coverage of the outermost speculated regions in all other speculation types [3].

In Figure 9 *load imbalance* and *coverage* are plotted, with the different speculation types clustered together for representative benchmarks [4]. Near-ideal TLS speedup is indicated with a high coverage score (red triangle) and a low imbalance (blue bar). Examining the data points of this graph for each of the speculation types against their respective speedup results in the previous sections can provide insight into performance limitations. Let us take *jpg2000dec*, for example. The low potential from innermost loops can be completely attributed to low coverage (*jpg2Kd IOloop* point in Figure 9). The significant increase in potential observed going from innermost loop speculation to OoO loop speculation is mostly because the coverage goes from 45% to nearly 100% (*OoOloop*). Method speculation does not provide any benefits for *jpg2000dec* and this is reflected in the relatively low coverage with increased load imbalance (*IOmethod* and *OoOmethod*). Mixed speculation, although showing near 100% coverage on speculated regions, has its performance potential hindered by the inclusion of imbalanced methods (*mixed*).

Some benchmarks have near 100% coverage, but do not achieve corresponding TLS speedups, even with perfect value prediction. This problem is caused by extreme *load imbalance*. As mentioned in Section 2.2.1, multiversioned caches can potentially alleviate the problem of load imbalance. To verify this, we simulated with 16 processors and 200 versions. Representative cases, featuring one benchmark from each benchmark suite, are shown in Figure 10. In some cases, multiversioning is successful in minimizing *load imbalance* and thus unlocking value prediction and, to a lesser extent, synchronization potential (*cjpeg IOmethod*, *cjpeg OoOmethod*, and *pmd IOmixed*). In these cases, however, we see no improvements in the baseline performance. This is due to data dependences between the speculative tasks. In the case of *pmd OoOmethod* multiversioning realizes performance improvement for base TLS, indicating the absence of data dependences. Finally, multiversioning is unable to provide any significant performance benefits for *gcc IOmethod* and *compress OoOmixed*. This indicates toward *load imbalance* that is inherent in the program structures chosen for task extraction. Task selection is key for improving performance in such cases.

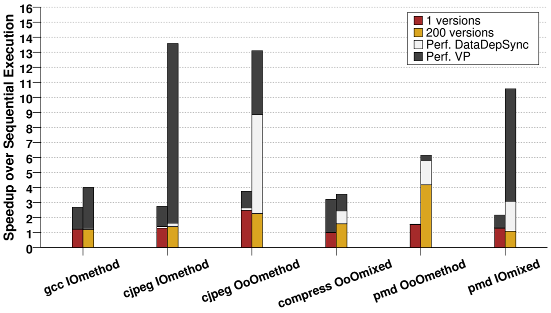

Figure 10: Performance impact of multiversioning for selected benchmarks and speculation types showing high coverage and high load imbalance, on 16 processors.

5 RELATED WORK

One of the first studies to attempt to establish performance upper-bounds of TLS in an ideal setting was [28]. In that study no compiler support particular to TLS was considered, and as such compiler optimizations, like induction variable elimination, were not considered. The work in [25], evaluated different design parameters such as speculation type (method and loop) and the order of spawning threads. A significant limitation of that work is that the spawning policies evaluated are considered independently and not concurrently, while they do not consider outer loops (a significant source of speedup). The work in [42] investigates the limits of method-level TLS and tries to establish the degree at which it varies between the imperative and

[3] In doing so we assume that the speculated regions can be partitioned into at least *n* parallel tasks, where *n* is the number of processors.

[4] The following subsets are formed based on load imbalance and coverage behavior: {*pmd*, *bloat*, *fop*, *db*, *jess*, *ray*}, {*jpg2Kd*, *jpg2Ke*}, {*cjpeg*, *djpeg*}, {*gcc*, *perlbench*, *sjeng*}, {*javac*, *jack*}, {*mpegaudio*, *antlr*}, {*calculix*, *sphinx3*}

the object-oriented programming models. Unfortunately, loop-level parallelism is omitted from that study and thus much of the potential of TLS is not exposed.

More recently, the work in [19] investigated the limits of performance improvements that may come exclusively from TLS and that could not be achieved through compiler auto-parallelization. In a follow-up study [18] the authors also investigated the effects of various threading overheads and misspeculation penalties. However, both studies assumed capabilities that are well beyond those of current auto-parallelizing compilers. At the same time, by not considering for speculative parallelization outer loops and not allowing out-of-order spawning of threads, they understated the potential of TLS systems. The work in [29] investigated again the potential of TLS, attempting to parallelize *all* loops that can benefit from TLS. Albeit closer to a better evaluation of TLS systems, that study only looked at applications from the SPEC 2000 and SPEC 2006 benchmark suites and, thus, does not have a fully representative mix of workloads. A different approach on the limits of TLS has been taken in [31]. In that study the authors manually parallelize several sequential applications and elaborate on the code transformations needed to achieve different levels of TLS performance. Although the insight provided into the TLS performance with the assumption of high-level programmer intervention is interesting, the authors do not study TLS-specific architecture optimizations.

The work in [42] is the only one that we are aware of that does experiment with dynamic applications. However, the benchmarks were compiled statically instead of using a Java runtime environment.

Compared to previous work, we perform an evaluation of TLS on a broad and diverse range of workloads. Also complementary to the related work is our investigation of *load imbalance* and *coverage* within the context of TLS as well as an evaluation of multiversioned caches as a means of improving load imbalance.

6 CONCLUSIONS

As multicore systems become common, the burden of extracting performance has shifted from the hardware toward the compiler/programmer side. Unfortunately, parallel programming is still hard and error prone and, perhaps more importantly, there is still a large sequential legacy code base. TLS systems offer a compelling alternative, in that they relieve the programmer from this burden by speculatively parallelizing sequential applications and dynamically checking whether the parallelization is correct.

In this paper we address many of the limitations of prior work and show that there is a lot to be gained from synchronization and value prediction. We also show that *load imbalance* and limited *coverage* are major factors in realizing potential along with *data dependences*. Task selection is, therefore, extremely important. We perform a limited task selection evaluation with perfect thread selection from nesting levels for loop level speculation, and employ a greedy task selection policy for OoO spawning but a lot remains to be evaluated and further exploration of task selection issues is left for future work.

We evaluate how individual high-level TLS architectural features contribute to overall performance gains, both in isolation and combination. Our results indicate that OoO spawning, with the greedy task selection policy employed, does not provide much benefit over in-order for most of the benchmarks.

REFERENCES

[1] Azul Systems. Vega 3 Processor. http://www.azulsystems.com.

[2] SPEC CPU2006. http://www.spec.org/cpu2006.

[3] SPEC JVM98. http://www.spec.org/jvm98.

[4] G. M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. *Spring Joint Computer Conf.*, pages 483–485, Apr 1967.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. “The DaCapo Benchmarks: Java Benchmarking Development and Analysis”. *Intl. Conf on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, October 2006.

[6] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. “Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun’s ROCK Processor”. *Intl. Symp. on Computer Architecture (ISCA)*, pages 484–495, June 2009.

[7] M. Chen and K. Olukotun. “Exploiting Method-Level Parallelism in Single-Threaded Java Programs”. *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 176–184, Oct 1998.

[8] M. Cintra, J. Martínez, and J. Torrellas. “Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors”. *Intl. Symp. on Computer Architecture (ISCA)*, pages 13–24, June 2000.

[9] M. Cintra and J. Torrellas. “Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors”. *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 43–54, February 2002.

[10] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry. “Tolerating Dependences Between Large Speculative Threads Via Sub-Threads”. *Intl. Symp. on Computer Architecture (ISCA)*, pages 216–226, June 2006.

[11] X. Dai, A. Zhai, W.-C. Hsu, and P.-C. Yew. “A General Compiler Framework for Speculative Optimizations Using Data Speculative Code Motion”. *Intl. Symp. on Code Generation and Optimization (CGO)*, pages 280–290, March 2005.

[12] J. Dou and M. Cintra. “Compiler Estimation of Load Imbalance Overhead in Speculative Parallelization”. *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 203–214, September 2004.

[13] 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”. *Conf. on Programming Language Design and Implementation (PLDI)*, pages 71–81, June 2004.

[14] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf. "MediaBench II Video: Expediting the Next Generation of Video Systems Research". *Microprocessors & Microsystems*, volume 33, pages 301–318, June 2009.

[15] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. "Speculative Versioning Cache". *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 195–205, February 1998.

[16] L. Hammond, M. Willey, and K. Olukotun. "Data Speculation Support for a Chip Multiprocessor". *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 58–69, October 1998.

[17] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. "Min-Cut Program Decomposition for Thread-Level Speculation". *Conf. on Programming Language Design and Implementation (PLDI)*, pages 59–70, June 2004.

[18] 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". *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 215–225, March 2007.

[19] 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". *Intl. Conf. on Supercomputing (ICS)*, pages 24–35, June 2006.

[20] S. W. Kim and R. Eigenmann. "The Structure of a Compiler for Explicit and Implicit Parallelism". *Intl. Wksp. on Languages and Compilers for Parallel Computing (LCPC)*, August 2001.

[21] V. Krishnan and J. Torrellas. "Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor". *Intl. Conf. on Supercomputing (ICS)*, pages 85–92, July 1998.

[22] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. "POSH: a TLS Compiler that Exploits Program Structure". *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 158–167, March 2006.

[23] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. "Simics: A Full System Simulation Platform". *IEEE Computer*, volume 35, pages 50–58, February 2002.

[24] P. Marcuello and A. González. "Clustered Speculative Multithreaded Processors". *Intl Conf. on Supercomputing (ICS)*, pages 77–84, June 1999.

[25] P. Marcuello and A. González. "A Quantitative Assessment of Thread-Level Speculation Techniques". *Intl. Parallel and Distributed Processing Symp. (IPDPS)*, pages 595–601, May 2000.

[26] P. Marcuello and A. González. "Thread-Spawning Schemes for Speculative Multithreading". *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 55–64, February 2002.

[27] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. "Dynamic Speculation and Synchronization of Data Dependence". *Intl. Symp. on Computer Architecture (ISCA)*, pages 181–193, June 1997.

[28] J. Oplinger, D. Heine, and M. Lam. "In Search of Speculative Thread-Level Parallelism". *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 303–313, October 1999.

[29] V. Packirisamy, A. Zhai, W. Hsu, P. Yew, and T. Ngai. "Exploring Speculative Parallelism in SPEC2006". *Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 77–88, April 2009.

[30] S. Pop, A. Cohen, and G. Silber. "Induction Variable Analysis with Delayed Abstractions". *Intl. Conf. on High Performance Embedded Architectures and Compilers (HiPEAC)*, pages 218–232. October 2005.

[31] M. K. Prabhu and K. Olukotun. "Exposing Speculative Thread Parallelism in SPEC2000". *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 142–152, June 2005.

[32] M. Prvulovic, M. Garzarán, L. Rauchwerger, and J. Torrellas. "Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization". *Intl Symp. on Computer Architecture (ISCA)*, pages 204–215, June 2001.

[33] C. G. Quinones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. "Mitosis Compiler: An Infrastructure for Speculative Threading Based on Pre-Computation Slices". *Conf. on Programming Language Design and Implementation (PLDI)*, pages 269–279, June 2005.

[34] 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". *Intl. Conf. on Supercomputing (ICS)*, pages 179–188, June 2005.

[35] V. Sarkar and J. Hennessy. "Partitioning Parallel Programs for Macro-Dataflow". *Conf. on LISP and Functional Programming (LFP)*, pages 202–211, 1986.

[36] G. Sohi, S. Breach, and T. Vijaykumar. "Multiscalar Processors". *Intl. Symp. on Computer Architecture (ISCA)*, pages 414–425, June 1995.

[37] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. "A Scalable Approach to Thread-Level Speculation". *Intl. Symp. on Computer Architecture (ISCA)*, pages 1–12, June 2000.

[38] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. "Improving Value Communication for Thread-Level Speculation". *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 65–75, February 2002.

[39] J. G. Steffan and T. C. Mowry. "The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization". *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 2–13, January 1998.

[40] T. N. Vijaykumar and G. S. Sohi. "Task Selection for a Multiscalar Processor". *Intl. Symp. on Microarchitecture (MICRO)*, pages 81–92, December 1998.

[41] M. Waliullah and P. Stenstrom. "Intermediate Checkpointing with Conflicting Access Prediction in Transactional Memory Systems". *Intl. Parallel and Distributed Processing Symp. (IPDPS)*, pages 1–11, April 2008.

[42] F. Warg and P. Stenstrom. "Limits on Speculative Module-Level Parallelism in Imperative and Object-Oriented Programs on CMP Platforms". *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 221–230, September 2001.

[43] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. "Compiler Optimization of Scalar Value Communication Between Speculative Threads". *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 171–183, October 2002.