Exhaustive Analysis of Thread-Level Speculation

Clark Verbrugge
Christopher J.F. Pickett
Alexander Krolik
McGill University, Canada
clump@cs.mcgill.ca cpicke@cs.mcgill.ca
alexander.krolik@mail.mcgill.ca

Allan Kielstra
IBM Toronto Lab, Canada
kielstra@ca.ibm.com

Abstract

Thread-level Speculation (TLS) is a technique for automatic parallelization. The complexity of even prototype implementations, however, limits the ability to explore and compare the wide variety of possible design choices, and also makes understanding performance characteristics difficult. In this work we build a general analytical model of the method-level variant of TLS which we can use for determining program speedup under a wide range of TLS designs. Our approach is exhaustive, and using either simple brute force or more efficient dynamic programming implementations we are able to show how performance is strongly limited by program structure, as well as core choices in speculation design, irrespective of and complementary to the impact of data-dependencies. These results provide new, high-level insight into where and how thread-level speculation can and should be applied in order to produce practical speedup.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming; D.4.8 [*Performance*]: Modeling and prediction

Keywords Parallelism, Thread Level Speculation, Performance

1. Introduction

Thread-level speculation (TLS) describes a range of approaches to automatic parallelization that attempt to take advantage of otherwise idle processors. It has been the subject of a large number of systems proposals and experimental studies based on novel hardware [13, 26], and more recently on pure software implementations [4, 17, 18, 21, 24].

SEPS'16, November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4641-2/16/11...
http://dx.doi.org/10.1145/3002125.3002127

Best performance in TLS systems depends on identifying code that can be profitably parallelized. Most designs focus on avoiding (or repairing) data-dependencies, which can cause *misspeculation,* and so reduce effective parallelism. Thread speculation, however, is resource constrained, and irrespective of succesfully avoiding data-dependencies, decisions to parallelize in one area affect the ability to parallelize later. The actual or potential speedup of a given piece of code under TLS is thus difficult to predict, often known only after the fact.

In this work we develop an abstract, flexible analytical model of TLS behaviour, based on the *method-level* (MLS) variation of TLS. Our approach is to perform a limit study, initially determining maximum possible performance irrespective of and orthogonal to the potential for misspeculation. Input timing traces (or estimates) of sequential execution can be fed into our model, and using an exhaustive analysis we can determine the best possible performance under different TLS design assumptions, including potential misspeculation. By separating concerns of how TLS responds to input program structure from how it responds to data-dependencies we are able to make progress in understanding the feedback complexity of TLS, providing further insight into why TLS does or does not perform well for a given program. Examination of the results of our analysis shows that strong dependencies exist between TLS design and program structure, that some TLS designs are better than others for certain coding practices, and reveals potential for future work that can exploit these differences.

Specific contributions of our work include:

- We define a general and expressive algorithmic abstraction of thread-based, method-level speculation. Our design allows for exhaustive, analytical exploration of behaviour, and includes a dynamic programming design for scalability well beyond trivial cases.
- We extend our base model with incremental complexity, representing three core forms of TLS: *in-order, out-of-order,* and *mixed* threading models. We show how to represent different parent/child signaling disciplines, and can

incorporate the representation of unsafe instructions, multiple forms of overhead, and the effect of misspeculation.

- We apply our formalism to several basic coding patterns (idioms), experimentally examining the interplay between overall TLS design, fork heuristics, code structure, and misspeculation. We show that even simple programming design differences can result in significantly different performance, independent of and with a comparable or larger impact than data-dependency considerations.

2. TLS Background

TLS achieves speedup by launching threads that operate on future program execution. At a *fork-point* a speculative thread is launched to begin executing at a future *join-point* in parallel with the initial, non-speculative thread. When the parent execution reaches the join-point it *signals* the speculative child thread to stop, *validates* it for correctness, and either *commits,* with execution continuing from wherever the child thread reached, or discards and *rolls back* the child with execution continuing from the join-point. The latter is called *misspeculation.* TLS variants exist that focus exclusively on loops, forking threads at the start of an iteration to execute the subsequent iteration [9], or methods as *method-level speculation* (MLS), forking at a call-site to execute the method continuation in parallel with the invocation [6]. *Arbitrary* speculation is also possible, forking speculative threads to execute any given future code sequence [3]. The latter is not necessarily more general, as all these forms of speculation can subsume each other with appropriate code transformations.

There are of course a number of safety and efficiency concerns in any TLS model. In traditional, *lazy* TLS designs, safety is provided through a combination of speculative isolation and validation. Isolation is provided by buffering speculative writes to ensure they do not conflict with parent reads or writes prior to join time. Parent writes may also affect child execution. Validation requires recording speculative reads, which can then be used at join time to verify that values read by a child match the state of memory reached by the parent at join time, and thus child execution correctly represents the behaviour that its parent would have followed at the join point. Other, recent designs have also explored *eager* speculation models, which allow all threads to directly read and write from main memory. Safety is then ensured by each thread maintaining a *shadow buffer* that records undo information, and a versioning table to record and identify correct read/write ordering. Eager designs eliminate the validation step, but have increased sensitivity to data dependencies and slower rollback. In this work we assume a lazy model, leaving the eager model for future work.

TLS implies a number of sources of overhead that can reduce performance. In an overall and approximate sense these costs can be aggregated into fork and join overhead, with the former including thread initialization, code-preparation and in the case of MLS *return-value prediction* [15], and the latter including signaling/termination, validation, and merge costs. Beyond basic overhead costs, the main limiting factor on potential speedup is imposed by the actual choice of fork points. To reduce misspeculation, these points must result in few data-dependency conflicts between parent and child threads. They should also include an appropriate balance of work within the method and its continuation, large enough that parallelization benefits exceed overhead concerns, but small enough that the probability of misspeculation does not grow too large [11]. Importantly, there are strong *feedback* concerns in forking heuristics—available cores are finite, and forking a thread at one point may preclude forking at a point in the near future, making the entire process extremely sensitive to the exact fork heuristic and program structure. It is the latter property that we focus on in this work.

3. Modeling TLS through MLS

Our model of TLS is based on the method-level (MLS) variation, which has the advantage of easily and syntactically identified fork and join-points. We assume a simple, stack-oriented program execution model consisting of sequential code interleaved with nested method calls. In order to model the control flow of MLS applied to such an execution we need to only identify calls, return points (continuations), and the base, sequential work performed. Note that we do not represent or track actual data-dependencies in this model: our primary goal is to examine the patterns of execution and parallelism generated within the combinations of program structure and MLS control flow, and in this sense misspeculation due to data-dependencies is primarily a source of additional overhead, reducing actual parallelism and system efficiency. We discuss misspeculation itself in section 3.3, and experimentally examine the impact in section 4.2.

3.1 Base Model

Our model builds on a sequential trace of *actions,* consisting of either method calls or basic work. Incorporating MLS involves adding in speculative thread forks (and joins), based on call-continuation pairings. This gives us a straightforward input representation we refer to as the *MLS constraint graph*, illustrated in figure 1 below.

The MLS constraint graph works in conjunction with a model of MLS execution. If we allow just one speculative thread, and assume joins are performed only upon termination, the potential behaviour is relatively easy to determine. Given the sequential execution trace described in figure 1, for example, the MLS system may choose to insert a single fork point before any call as the non-speculative thread executes. All possible resulting execution sequences are shown in figure 2.

Note that we can already observe in this simple execution context that the parallelism generated strongly depends on

```
a) A() {          B() {          C() {
     work1          work3          work5
     B()            C()          }
     work2          work4
   }              }

b) A → w1 → B → w3 → C → w5 → w4 → w2 → •
```

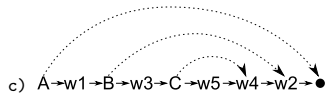

Figure 1. a) Code, b) an execution sequence due to executing "`A();`" and terminating at •, and c) the corresponding MLS constraint graph; dashed edges are continuation edges.

```
(1) ; (A→w1→B→w3→C→w5→w4→w2) | (•)
(2) A→w1 ; (B→w3→C→w5→w4) | (w2→ •)
(3) A→w1→B→w3 ; (C→w5) | (w4→w2→ •)
(4) A→w1→B→w3→C→w5→w4→w2→ •
```

Figure 2. Possible MLS execution sequences for the code in figure 1. The fork point is shown by a ';' and is followed by a parallel computation separated by a '|'.

the specific forking choices made. Sequence (1) achieves no parallelism but does have speculative overhead. Sequences (2) and (3) have some parallel execution, but have different degrees of balance between threads. Sequence (4) follows if no fork point is selected, and is just sequential execution.

In this design, each potential MLS execution consists of three main sections. An execution consists of a sequential *preamble* terminating in a fork and method-call (or program end in the trivial case). A fork point divides subsequent execution into 2 pieces: a (non-speculative) parent thread that executes until just before the continuation point, and a (speculative) child-thread that executes all code from the continuation onward. That is, our original sequential execution can be "parsed" into an MLS execution:

$$\text{preamble}(S)\ ;\ \text{non-spec}(A)\mid\text{speculative}(B)$$

The process for discovering all MLS executions is then straightforward. We incrementally grow the preamble S. If we encounter a potential fork point we consider 2 options, one where we launch a speculative thread and split the execution into A and B, and one where we do not and just continue growing the preamble.

Calculating parallel speedup in this model is analytically trivial. Given a base sequential sequence $t_1, \ldots, t_n$ the time taken can be calculated (simply) by summing the weight (ω) of each individual operation. Time taken by a sequence containing a fork is calculated (in general) recursively, considering the overlap of parent and child executions, as well as a fork cost (F) and a join cost (J). This gives us the following definitions for a time calculation function τ which we apply to both sequential and forked code.

$$\tau(t_1, \ldots, t_n) = \sum_{i=1}^{n} \omega(t_i)$$

$$\tau(S;A|B) = \tau(S) + F + \max(\tau(A), \tau(B)) + J$$

3.2 Multiple speculative threads

Most speculative systems allow multiple speculative threads, taking advantage of as many of the available CPUs as possible to improve parallelism. Three main ways exist to extend a basic 2-thread MLS system, *out-of-order*, *in-order*, and *mixed* nesting.

In the *out-of-order* model a non-speculative parent thread may create multiple children as it descends down a call chain. Thus a single non-speculative thread can have many speculative children, although speculative threads do not have further speculative children. An example is shown in figure 3; here out-of-order parallelism helps significantly in improving parallelism.

```
a) A() {          B() {          C() {
     B()            C()            work3
     work1          work2        }
   }              }

b) A → B → C → w3 → w2 → w1 → •
c) ; (A → ; (B → ; (C → w3) | w2) | w1) | •
```

Figure 3. a) Code, b) sequential execution sequence due to executing "`A();`", and c) an out-of-order MLS execution assuming an arbitrary number of threads (CPUs) available.

An alternative design is to allow speculative children to themselves launch speculative children. This is known as *in-order* nesting, wherein each thread, speculative or not, may have at most one speculative child. Conceptually, in-order speculation tends to perform well in situations where out-of-order nesting does not, and vice versa. An example of in-order nesting is shown in figure 4. Note that out-of-order nesting would result in less possible parallelism here, since the lack of nested calls in any preamble means that at best a single speculative thread could be forked.

```
a) A() {          B() {          C() {
     work1          work2          work3
   }              }              }

b) A → w1 → B → w2 → C → w3 → •
c) ; (A → w1) | (; (B → w2) | (; (C → w3) | •))
```

Figure 4. a) Code, b) sequential execution sequence due to executing "`A();B();C()`", and c) an in-order MLS execution assuming an arbitrary number of threads (CPUs) available.

Finally, one may of course combine out-of-order and in-order techniques, allowing each thread to have any number of speculative children, whether the parent thread is speculative or not. This is *mixed* nesting. An example is shown in figure 5.

Although it is more difficult to see, mixed nesting results in an maximal parallelism, parallelizing both calls to $B()$ and to $C()$ in our example. This can be contrasted with out-of-order and in-order designs, which parallelize along only one major branch of the computation in each case.

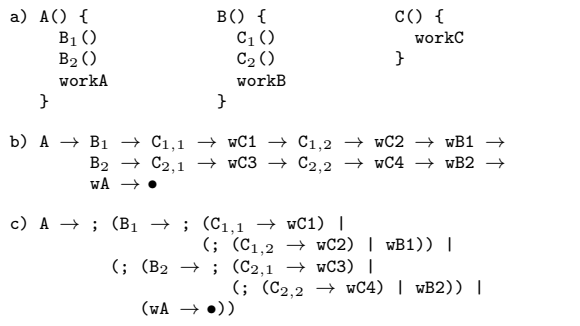

Figure 5. a) Code, b) sequential execution sequence due to executing "`A();`", and c) a mixed nesting MLS execution.

3.3 Signaling, Joining, Stopping, and Misspeculation

To complete our model we now incorporate the idea of limited thread resources, redeploying a CPU once a speculative execution has been joined or otherwise terminated. To accommodate this kind of behaviour we need to explicitly recognize the point at which threads are signaled to stop execution and prepare for joining. Two main approaches to thread joining are possible and can be represented in our system, which we denote *forward-signaling* and *backward-signaling*. We address both below, as well as two factors that reduce speculative performance: *unsafe* instructions, (which prevent speculation from proceeding further), and misspeculation (which rolls back speculative execution).

In **forward-signaling**, once a parent thread reaches the execution point at which its child began execution it signals the child to stop, joins it, and then proceeds having recovered the speculative thread resource. This enables reuse of the child resource (CPU) for further speculation in subsequent execution, although only within whatever remains of the continuation that was executed by the child.

Forward-signaling applies most naturally to out-of-order execution, but unfortunately is not as effective for in-order execution. With in-order nesting, a long but potentially parallelizable parent execution will not be exploited since the parent thread must complete all of its work before it reaches the join point and is able to recover the speculative child-thread. In-order models benefit instead from *backward-signaling,* wherein signaling roles are reversed to allow the parent thread to receive "advance notice" of terminated speculative child-threads.

Backward-signaling is performed when a speculative child which has terminated signals its parent. The parent thread can then store the child state for later joining, and reuse the speculative thread resource to launch another speculative child prior to joining. As with forward-signaling, backward-signaling enables more speculation with fewer CPUs, differing in that parent threads may not launch more children until a child has terminated.

Backward-signaling has the disadvantage that terminated speculative thread states need to be retained until parent execution reaches the corresponding continuations. This requires the addition of non-trivial memory management for storage and retrieval of isolating buffers and speculative stack frames, and so is less common, particularly in hardware models where thread buffering is done through dedicated cache partitioning. In extending our model we thus focus mainly on forward-signaling.

We incorporate a forward-signaling joining procedure by extending our MLS representation. Instead of just $S;A|B$, we allow the execution of B to be truncated, splitting B into two pieces, the code executed prior to the signal, and the code executed after the parent joins with its child. The latter code is then evaluated recursively adding back in the recovered speculative thread resource. As a general template then, we model MLS execution of a sequence as a recursive decomposition of a sequential sequence into $S;A|B+C$. S is the sequential preamble ending in a method call, A is the method body, B is the continuation up to the point at which the speculative thread is joined, and C is the remaining execution, giving us the following overview equation. Assume $T = SABC$, then:

$$\text{MLS}(T) \;=\; S \;;\; \text{MLS}(A) \mid \text{MLS}(B) \;+\; \text{MLS}(C)$$

Unsafe instructions are instructions which may not be executed safely in a speculative context. These typically include I/O, synchronization, and any other instructions that may have a global effect not completely captured and made reversible by buffering basic reads and writes. An unsafe instruction is easily modeled within the same abstraction; if a speculative thread encounters an unsafe instruction execution is stopped, and the speculative thread waits to be joined there.

Misspeculation. The existence of data-dependencies is of course a major concern for actual speculative performance. A read by a speculative thread of data later written by a parent thread that is logically earlier in execution can result in misspeculation—the speculative execution fails to validate, and is aborted instead of joined. This behaviour has two main impacts. Most simply, misspeculation implies additional overhead in the system: thread creation and joining of a misspeculating thread is wasted effort, since the misspeculating execution must be discarded and the code re-executed. A further, more subtle impact is due to the reduction in thread resources and potential parallelism. A misspeculating thread is still a dedicated resource, and thus is not available for other speculative purposes for the duration of its failed execution.

3.4 Exhaustive Algorithm

Figure 6 formalizes and summarizes the notions discussed in this section. Given a sequential execution, it expresses all possible in-order, out-of-order, or mixed MLS executions, accommodating forward-signaling, non-speculative instructions, misspeculation, and limited thread resources.

```
Let T = t_1, t_2, ..., t_n be a sequential trace of actions.
MLS(T,σ,time) =
  for all S = preamble(T, σ) s.t. τ(S) ≤ time
    let (t_{|S|+1},t_b) be a continuation edge
    T_A = t_{|S|+1}, ..., t_{b-1}
    for all σ_1, σ_2 = σ-1,0      // for out-of-order
                       0,σ-1      // for in-order
                       split(σ-1) // for mixed
      for all A = MLS(T_A,σ_1,time-τ(S)-F)
        T_B = t_b, ..., t_n
        for all B = MLS(T_B,σ_2,τ(A))
          t_f = misspec(B) ? t_b : t_{|S|+|A|+|B|+1
          T_C = t_f, ..., t_n
          τ(S; A|B) = τ(S) + F + max(τ(A), τ(B)) + J
          for all C = MLS(T_C,σ,time-τ(S; A|B))
            τ(S; A|B + C) = τ(S; A|B) + τ(C)
            return S ; A | B + C
```

Figure 6. Algorithm for enumerating in-order, out-of-order, or mixed MLS executions, with forward-signaling and a bounded number of threads. T is the input trace of actions, σ the number of speculative threads that are available for allocation, and time is the maximum time before a parent signal will occur. The preamble(T,σ) function returns T and if $\sigma > 0$ then all prefixes of T that end before a method call (fork point) as well. The split($\sigma - 1$) function returns all non-negative pairs σ_1, σ_2 such that $\sigma_1 + \sigma_2 = \sigma - 1$. The $\tau()$ function returns the time used by the given sequence.

The process begins by providing an input consisting of the sequential sequence to decompose (T), a number of available speculative threads (σ), and a timeout for when the execution will be joined—for initial (top-level) input the timeout is infinite, as the non-speculative thread is not joined. The MLS function then returns all possible MLS executions of that sequence. The function initially and optimistically tries to decompose T into $S; A|B$, splitting off C and creating $S; A|B + C$ instead only if necessary.

Within the function, all possible preambles (up to the timeout limit) are considered, each of which is assumed to terminate in a method call (if not then the result is just a single, sequential execution of T). The method call defines the split between the preamble S, the method body A, and its continuation B. Once that split point is established, a speculative thread will be in use to execute B, and the remaining threads are allocated to the recursive decompositions of A and B. In the case of out-of-order all threads go to A, for in-order threads go to B, and for mixed all possible splits of the thread resources must be considered.

Recursive decompositions of A are then computed given the input timeout, subtracting the time consumed by the preamble and the forking itself. Since B can only execute until joined, its timeout is given by the duration of the recursive execution of (a given) A. Joining prior to the completion of all B (with a slight abuse of notation) splits B into BC, with C being the remaining execution. Normally, in the absence of misspeculation (the misspec function returns false), the starting point of C is the code after A and B join. If misspeculation of B occurs (misspec returns true) the execution of B is discarded, and the starting point of C is moved back to include all of B in its execution. The overhead of creating and joining the misspeculating thread is thus incorporated, as well as the impact on A of having reduced thread resource due to their use within the misspeculating execution. In either case, since C executes after the join it has available the full thread resources, and whatever time remains after $S; A|B$ (any code of C still remaining after timeout is left unexecuted, and becomes the C part of the parent, recursive execution). For assessing speedup, the total time taken is calculated and associated with the input sequence.

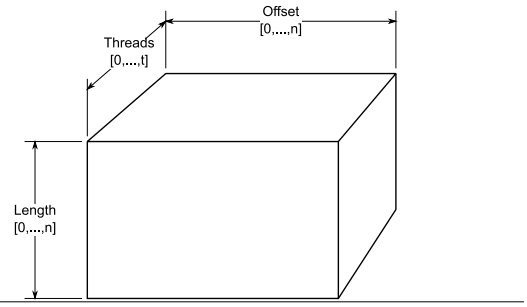

Figure 7. Dynamic programming model for finding optimal fork-points.

Note that we can extend this model to accommodate other variations on TLS as well. Backward-signaling, for instance, can be incorporated by inverting the order in which we determine the available timeouts. That is, we simply need to reverse the dependency between A and B, evaluating B first and passing the elapsed time as a minimum timeout *before* a thread could be launched to the recursive evaluation of A. A *bi-directional* signaling model is also possible, but would require an additional, non-trivial recursion to balance the time consumed by A given the resources passed back from B, the amount of which recursively depends on the time consumed by A.

3.5 Dynamic Programming Model

Our exhaustive algorithm involves multiple, nested recursion to generate all possible executions. If we are mainly interested in finding a single optimal execution, a dynamic programming approach can be used to cache the many recursive invocations used in the algorithm. Figure 7 shows the resulting abstract space. We fill the discrete cells of this cube layer by layer, growing the size of execution trace fragments we consider until we cover the entire execution. Each layer determines optimal execution time for a fixed length of trace fragments starting at all possible offsets, given all possible thread resource assumptions. The result of recursive calls in the algorithm of figure 6 are thus already available in lower

Name	Description
iter	A sequence of 10 calls to the same work function.
head	Head-recursion, 10 levels deep, each call executing a work function upon return.
tail	Tail-recursion, 10 levels deep, each call executing a work function before the recursive call.
tree	A recursive, post-order computation within a binary tree down to 3 levels (7 units of work total).

Table 1. Synthetic benchmark suite. Note that these represent control-flow abstractions only, and do not include data-dependencies.

layers, and while it still involves non-trivial computation at each point analysis time is improved by orders of magnitude.

4. Experimental Analysis

An experimental investigation is performed by applying the algorithm of figure 6 to different program fragments and evaluating speedup, parametrization impact, and the relative cost of misspeculation. As a benchmark suite we have initially concentrated on small program traces based on highly general, ubiquitous coding idioms, as summarized in figure 1. These code fragments have been chosen as representative forms of commonly used, repetitive control-flow of the form typically targetted (at a high level) in TLS optimization. We focus on these idioms rather than actual traces of program fragments in order to let us verify the behaviour through inspection, and as they are already sufficient to demonstrate a variety of interesting behaviours under MLS.

We also assume a very simple model of execution and overhead costs: method-calls take 5 units, forks 5 units, joins take 20 work units each, and actual work execution takes 1000 units. The cost of calling and the thread fork/join operations are chosen to roughly match the assumed cycle-cost of similar operations in typical TLS hardware simulations [22], and the work-weight is chosen to be much larger in proportion. Our experiments show minor, proportional differences from different overhead parametrizations, but the character of results is quite insensitive to these choices.

4.1 Speedup

Different choices of how and when threads are forked are expected to impact the final performance. We thus analyze the three basic MLS models (in-order, out-of-order, and mixed) under a range of thread resources (from 1 to 9 speculative threads available), and measure the maximal speedup possible under any forking strategy, the speedup obtained by a "greedy" forking heuristic (choosing to fork at a method call if a CPU is idle), and an "average" speedup over all possibilities. Maximal speedup provides a theoretical optimum that limits any fork heuristic, averaging is meant to provide a baseline showing behaviour when no effort is made to develop an effective fork heuristic, while greedy represents a straightforward, but still reasonable fork heuristic. Results for the greedy option under mixed show the maximum speedup possible for any possible thread division.

Also note that in our model, units of execution (trace symbols) are either executed or not—even with signaling we do not split work units when a signal occurs, and assume that a signal occurring within a work-unit is not acted upon until the work is completed by the speculative thread. Although this limits how work can be partitioned, it also more accurately models the common practice of using infrequent polling (eg on method entries, exits, and backward loop branches) instead of true asynchronous signaling for inter-thread communication.

Results are shown in figure 8. In terms of maximal possible performance, striking differences are evident in how the thread models respond to each of the different benchmark structures. An in-order approach is generally more effective than out-of-order, and this can be understood from how the strategies interact with the benchmark structure. In the case of iteration and tail recursion, in-order performs better than out-of-order since subsequent iterations (or recursive calls) are essentially always contained in the continuation of the current iteration (call). Head recursion allows both strategies to be effective since out-of-order can launch threads as the recursion descends, while in-order can be effectively applied once the recursion bottoms out. The mixed model, unsurprisingly, is able to combine and sometimes exceed the benefit of either in-order or out-of-order alone. This is most apparent in tree, where the pure strategies are limited to exploiting one branch down the tree while a mixed approach lets the best strategy be selected at each branch in the descent.

Average performance is mainly interesting in providing evidence of the extent of bias toward suboptimal performance. The low average behaviour suggests the bulk of fork strategies do not provide much speedup, and good performance is only found by applying some effort to identify the few, best forking choices.

In many cases, however, greedy behaviour turns out to be effective at finding these better fork points, although this too depends on the MLS design. In the case of iter, greediness is optimal irrespective of the MLS model. In head, tail, and tree greedy works well for out-of-order and of course mixed, but quite poorly for in-order. For in-order, a simple greedy approach tends to fail due to the fact that there is a single method call entry point to all these tests—launching threads for the continuation has little to no impact on the bulk of the work. Backward or bidirectional-signaling would improve this, repurposing speculative threads after completion but before joining, but it also illustrates the importance of matching speculative design to the code structure.

4.2 Misspeculation

The presence of data-dependency induced misspeculation is of importance to TLS [6], and has inspired a wealth of designs to mitigate the cost, both deterministic [2, 25], and

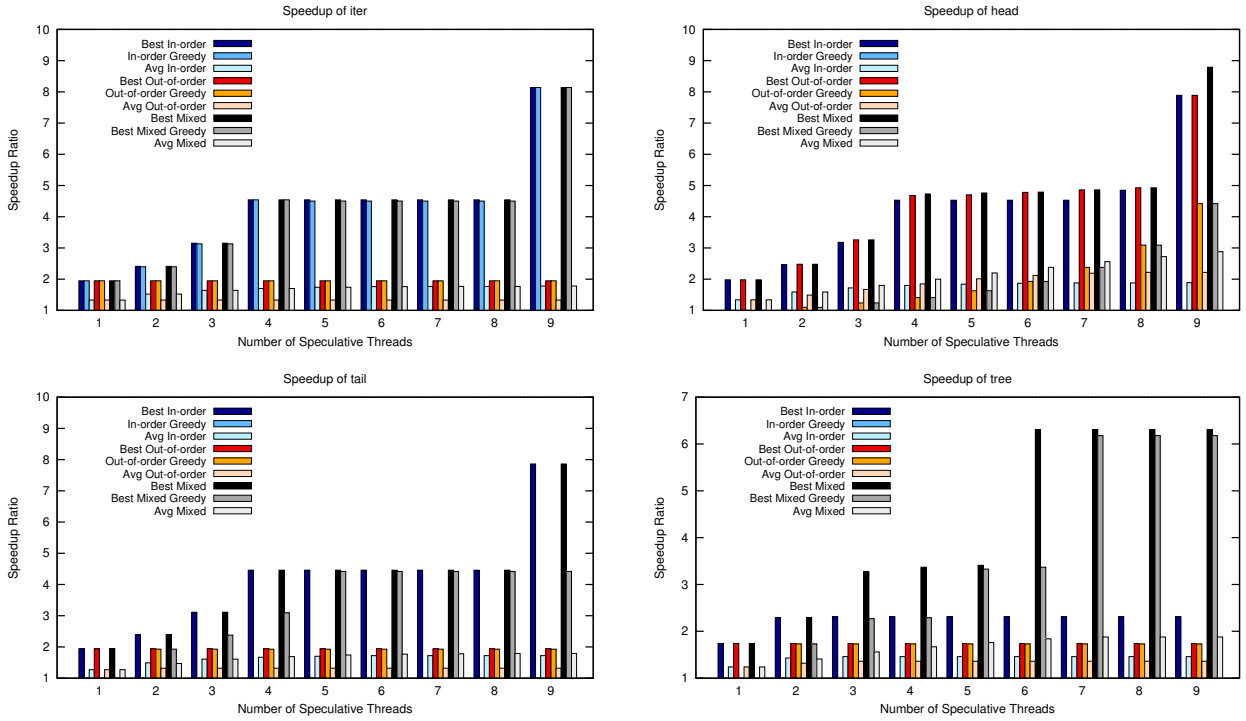

Figure 8. Speedup for benchmarks given different maximal thread resources, thread models, and fork heuristics. A bar is shown for each number of available speculative threads; maximal speedup, greedy speedup, and average speedup are grouped and shown for in-order, then out-of-order, and finally mixed speculative strategies respectively. Maximum theoretical speedup is 10, except for tree which is 7.

heuristic [8]. Given the significant impact of speculation style and code structure on TLS performance, however, it is important to quantify the relative effects.

Methodically examining misspeculation within synthetic benchmarks has interesting complexities. Our model allows us to induce misspeculation on any call-site, but if we naively continue to measure best-case performance then we end up merely considering situations where speculation at that call-site is simply avoided altogether—this tends to have a detrimental impact on performance, but does not fully represent the cost of misspeculation itself. Alternatively, if we force misspeculation at a call-site we are also necessarily forcing speculation at that call-site, which in general may not be an optimal choice.

To analyze the potential impact of misspeculation on performance, we conducted experiments for various misspeculation behaviours. Each misspeculation behaviour was considered with two measures; one wherein actual misspeculation is indeed forced, and one wherein the call-site will misspeculate if speculated upon, and thus is in most cases skipped. The latter can also be viewed as the steady-state response of an adaptive system which learns that a site misspeculates and so learns to avoid it.

Misspeculation has a large impact on speedup of course. We do not show numerical results, however, both for space reasons and as the results can be easily interpreted analytically. A single, forced misspeculation, for instance, generally cuts speedup by half, independently of the misspeculation location. This is, however, a natural consequence of how speedup is calculated. There are N units of work, optimally executed concurrently for a total parallel time of N/t, and thus a speedup of t. If one thread's work is re-executed then total time is increased by N/t, reducing speedup to $t/2$. In general, with m misspeculations, we expect speedup to be reduced to $t/(m+1)$. Note that this behaviour is in some sense worst-case, and exposed by the high degree of parallelism the benchmarks are otherwise able to achieve—if non-speculative execution is significantly longer than speculative execution, and so speedup is already non-optimal, it may sometimes be possible for the misspeculation of a grand-child thread to occur within an otherwise idle speculative thread, and thus not to have as large a relative impact on overall speedup. For benchmarks with less parallelism, the impact on speedup is lower.

Within all our misspeculation experiments, an important observation is with respect to the relative scale of degradation (or improvement). While misspeculation has an important and significant impact, it is not necessarily larger than the variations shown earlier due to the interaction between speculative design and code structure. Effective speculation is best achieved by considering both these aspects, and neither a focus purely on misspeculation nor on design/code is sufficient as a general, optimal solution.

4.3 Scalability

Our basic exhaustive analysis generates all possible traces, and is thus limited in scalability. When only the optimal solution is interesting, however, such as in potential implementations of MLS, the dynamic programming solution given in section 3.5 offers much better performance and is thus more practical for analyzing larger scale traces and systems. Figure 9 summarizes scalability results for the analysis itself.

Differences in the length of the input trace have the most important effect on the performance of the algorithm. We are able to analyze an NQueens problem of size 7 in reasonable time, a respectable result, although the rapid growth in analysis time clearly limits the size of traces that can be analyzed to short segments, or requiring coarser granularity.

The right side of figure 9 shows that the algorithm performance is also directly affected by the system resources, in that a larger number of threads is more costly to explore, although the impact is much less dramatic. For both in-order and out-of-order speculation modes the execution time is linear in the number of threads, and even for mixed mode, where all thread allocations are considered, the execution time scales quadratically.

5. Related Work

A wide variety of TLS [13] (and MLS [6]) approaches have been defined, in most cases supporting unique variants of out-of-order, in-order, or different forms of nested threading models. Research has concentrated on hardware and hybrid hardware/software designs [26], primarily as a means of ensuring low overhead and maximizing potential speedup. Pure software approaches to TLS are less common, but have also been explored [20, 21]; fine-grain speculation and short thread-lengths, however, can easily lead to relatively large overhead concerns. More recently, Ding *et al.* proposed coarse-grain, software-based *Behavior Oriented Parallelism,* which uses the virtual memory system to isolate "possibly-parallel" regions [10]. This design allows for overhead concerns to be hidden by larger scale parallelism, and the authors show factor-of-2 speedups on several realistic, originally sequential benchmarks.

Yiapanis *et al.* have also proposed software speculative systems, *MiniTLS* and *Lector*, that have reduced speculative overhead compared with traditional systems [31]. Focusing on small numbers of threads, their systems use an optimized structure for storing speculative data that allows the "eager" implementation to have minimal rollback time and the "lazy" variant to have quick data dependence checks. In a more recent work, Yiapanis *et al.* summarize the considerations when designing a software TLS system (violations, thread scheduling, data visibility, etc.) and the recent developments in these areas [32].

Other work on software speculative systems has explored optimizing existing systems for specific contexts. Martinsen *et al.* researched the effect of speculative parameter tuning on a previously implemented JavaScript engine [17]. By limiting the speculative depth and thread count, web applications can improve speculative performance while reducing the system's memory usage.

Whatever the threading model, determining where and when to fork threads is one of the fundamental challenges of TLS systems. As well as the basic safety problem of avoiding or repairing data-dependencies, to show speedup it is necessary that the amount of work exceeds any actual overhead, and thus the "length" or duration of speculative threads is recognized as an important heuristic criterion. Warg and Stenström explore this behaviour in an MLS system and show that a simple "last-value" predictor (applied to thread length) can be a very effective way of ensuring this property, eliminating a large amount of unnecessary overhead from lack of actual parallelism [28]. Other work on fork heuristics has shown that a careful balance must be achieved in heuristic choices—applied too conservatively, fork heuristics can lead to significant under-speculation, also reducing performance [29]. The recent *POSH* system uses several optimizations as part of fork (task refinement) heuristics, considering thread-length, dependency and profiler information [16]. Their system requires tasks be spawned in reverse execution order, imposing an out-of-order speculation model. Simulation results with this design show an average 1.3 speedup on SpecInt benchmarks, the same behaviour others have reported with optimized out-of-order designs [23].

TLS can also be combined with other techniques for increased performance. Xekalakis *et al.* explore combining TLS with "HelperThreads", "RunAhead" and "MultiPath" execution to improve instruction level parallelism [30]. Using the *SESC* simulator, experiments show a mixed execution model reduces L2 cache misses, pipeline flushes and improves ILP, the primary cause of increased performance.

Abstract models of parallel execution have been of interest for some time. Many have been developed in the context of pure or partial functional languages, where dependency requirements are simplified. An early approach was given by Greiner and Blelloch, defining a *parallel speculative λ-calculus* to model "call-by-speculation," an approach to parallelism wherein function arguments are evaluated concurrently with the function itself [14]. Their concern is in further parallelizing initial designs that serialize behaviour within a queueing model typically used to block threads accessing the same, but unavailable argument data. Provable time efficiency is then demonstrated within a λ-calculus implementation. Baker-Finch *et al.*, develop a detailed operational semantics for an extended λ-calculus representing GPH, a parallel version of the Haskell language with lazy evaluation [1]. Their design allows for expression of control-based parallelism based on the *par* annotation, although it could be extended to implicit and fully speculative models.

Our approach in this work is partly inspired by previous work done by Oplinger *et al.*, examining behaviour of

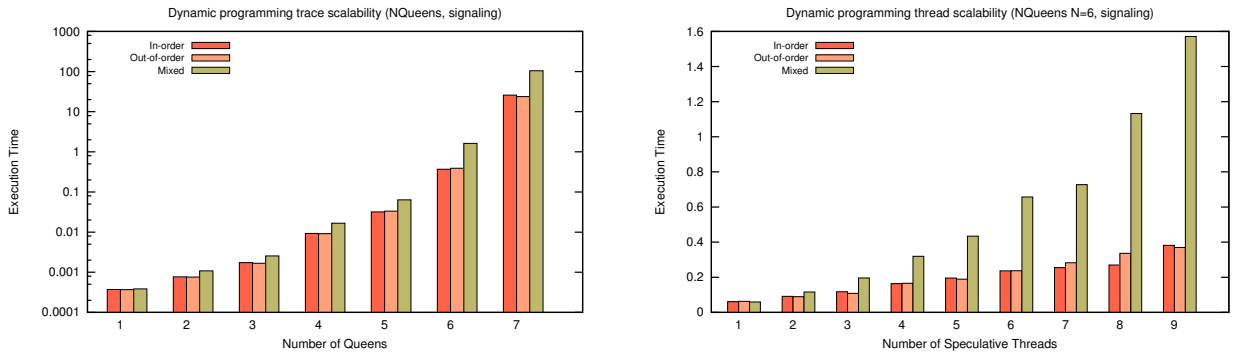

Figure 9. Performance of the dynamic programming algorithm, assuming join points at every call. Left varies the trace size of NQueens with 9 speculative threads, and right the number of speculative threads on a fixed NQueens (N=6) trace. Time in *s*, using a Java 1.6 implementation on a 2.00GHz Xeon.

an abstract "greedy" TLS thread model, either always forking threads or (in the case of a bounded number of threads) using heuristic thread-priorities to model scheduling concerns [19]. They also use a trace-based analysis, abstracting many overhead and machine details to determine optimal performance, at least under their fork and scheduling assumptions. Although their design considers only one threading behaviour and is not an exhaustive exploration, this allows for some important conclusions to be made about structure, and in particular they are able to show that both loop and procedure-based parallelism are necessary to best exploit the potential parallelism of realistic applications. This is a position also argued in more recent experimental work by other researchers [16].

The abstraction we investigate here does not form an explicit taxonomy, but instead builds on previous work on modeling and understanding MLS [20]. This earlier work does not perform exhaustive analysis, aiming more at developing a stack model for MLS execution with corresponding visualization, but has inspired our basic approach, and we have used several of the code idioms extracted in the course of that work in the context of our investigation.

With less abstraction, detailed performance models have also been defined, with the majority of attention devoted to improving loop-based speculation. As an extension to their Hydra design, Chen and Olukotun's *TEST* system defines hardware-based support for estimating the performance of different thread decompositions [5]. This is applied during runtime to help identify loops appropriate for TLS execution, allowing the rest of the *Jrpm* hardware-software hybrid system [7] to then recompile the corresponding method to take advantage of speculative hardware. The TEST system considers iteration dependencies, as well as lower-level considerations such as the potential for buffer overflows.

There are many ways to approach and estimate the potential of loop-level speculation. Du *et al.* also define a cost metric, using a data-dependence graph annotated with probabilities to estimate the cost of misspeculation [12]. They use this to locate minimal cost candidates suitable for TLS. Wang *et al.* build a loop graph, modeling the nesting relation between loops within a program, and use this in conjunction with coverage and individual loop speedup estimates to compute a heuristically optimal selection of loops upon which to apply loop-level TLS [27].

Dou and Cintra take a more comprehensive approach, incorporating thread sizes as well as branch probabilities and TLS overheads, in order to form "tuples" describing different combinations of all possible executions of a loop body, from which a minimal execution set can be extracted [11]. To maintain practicality within a compiler framework, they do not consider nested loops or recursion. Interestingly, even with this intricate model simulation results show a broad range of speedups and slowdowns depending on the benchmark, again emphasizing the need to better understand how choices in parallelization interact with program structure.

6. Conclusions and Future Work

Our work complements and extends the many existing efforts that concentrate primarily on ameliorating the impact of data-dependencies in TLS systems. We have shown that a deep and more holistic understanding of code structure is a further, essential property of MLS performance that must be considered to achieve reliable and practical speedup. Using an exhaustive exploration applied to common code structure idioms we demonstrate the large impact code structure has on potential speedup, and show how structure, speculative design, and fork choices can interact to drastically alter performance, as much as or more than misspeculation.

There are many interesting aspects of TLS we can further explore with our model. A detailed, but still abstract model of data dependencies would allow for finer-grain analysis of misspeculation, and enable us to abstractly evaluate and compare the impact of various techniques that either stall at dependencies or aim to reduce critical path lengths. We would also like to extend our model to accommodate bidirectional-signaling and different inheritance

models. Our main interest, however, is scaling up the design and combining it with models of expected control flow to dynamically select an appropriate speculation strategy in the context of a complete MLS prototype.

Acknowledgments

Thanks to the IBM Centre for Advanced Studies and the Natural Sciences and Engineering Research Council of Canada.

References

[1] C. Baker-Finch, D. J. King, and P. Trinder. An operational semantics for parallel lazy evaluation. In *ICFP '00*, pages 162–173, 2000.

[2] S. Balakrishnan and G. S. Sohi. Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs. In *ISCA'06*, pages 302–313, June 2006.

[3] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *SPAA '02*, pages 99–108, Aug. 2002.

[4] Z. Cao and C. Verbrugge. Mixed model universal software thread-level speculation. In *ICPP '03*, pages 651–660. IEEE, Oct 2013.

[5] M. Chen and K. Olukotun. TEST: a tracer for extracting speculative threads. In *CGO '03*, pages 301–312, 2003.

[6] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *PACT'98*, pages 176–184, Oct. 1998.

[7] M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *ISCA'03*, pages 434–446, June 2003.

[8] P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *PPoPP'03*, pages 25–36, June 2003.

[9] M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP '03*, pages 13–24, June 2003.

[10] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI '07*, pages 223–234, 2007.

[11] J. Dou and M. Cintra. A compiler cost model for speculative parallelization. *TACO*, 4, June 2007.

[12] 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 *PLDI '04*, pages 71–81, 2004.

[13] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin–Madison, Madison, Wisconsin, USA, 1993.

[14] J. Greiner and G. E. Blelloch. A provably time-efficient parallel implementation of full speculation. *TOPLAS*, 21(2): 240–285, 1999.

[15] S. Hu, R. Bhargava, and L. K. John. The role of return value prediction in exploiting speculative method-level parallelism. *JILP*, 5:1–21, Nov. 2003.

[16] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: a TLS compiler that exploits program structure. In *PPoPP '06*, pages 158–167, 2006.

[17] J. K. Martinsen, H. Grahn, and A. Isberg. The effects of parameter tuning in software thread-level speculation in javascript engines. *TACO*, 11(4):46:1–46:25, Jan. 2015.

[18] C. E. Oancea, A. Mycroft, and T. Harris. A lightweight in-place implementation for software thread-level speculation. In *SPAA'09*, pages 223–232, Aug. 2009.

[19] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *PACT'99*, pages 303–313, Oct. 1999.

[20] C. J. F. Pickett. *Software Method Level Speculation for Java*. PhD thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, Apr. 2012.

[21] C. J. F. Pickett and C. Verbrugge. SableSpMT: A software framework for analysing speculative multithreading in Java. In *PASTE'05*, pages 59–66, Sept. 2005.

[22] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *PLDI '05*, pages 269–279, 2005.

[23] 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 *ICS'05*, pages 179–188, June 2005.

[24] P. Rundberg and P. Stenström. An all-software thread-level data dependence speculation system for multiprocessors. *JILP*, 3:1–28, Oct. 2001.

[25] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *HPCA '02*, pages 65–75, 2002.

[26] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, Aug. 2005.

[27] S. Wang, X. Dai, K. Yellajyosula, A. Zhai, and P.-C. Yew. Loop selection for thread-level speculation. In *LCPC'05*, volume 4339 of *LNCS*, pages 289–303, 2006.

[28] F. Warg and P. Stenström. Improving speculative thread-level parallelism through module run-length prediction. In *IPDPS '03*, pages 12.2–, 2003.

[29] J. Whaley and C. Kozyrakis. Heuristics for profile-driven method-level speculative parallelization. In *ICPP'05*, pages 147–156, June 2005.

[30] P. Xekalakis, N. Ioannou, and M. Cintra. Mixed speculative multithreaded execution models. *TACO*, 9(3):18:1–18:26, Oct. 2012.

[31] P. Yiapanis, D. Rosas-Ham, G. Brown, and M. Luján. Optimizing software runtime systems for speculative parallelization. *TACO*, 9(4):39:1–39:27, Jan. 2013.

[32] P. Yiapanis, G. Brown, and M. Luján. Compiler-driven software speculation for thread-level parallelism. *TOPLAS*, 38(2): 5:1–5:45, Dec. 2015.