Abstract Analysis of Method-Level Speculation

Sable Technical Report No. 2011-3

Clark Verbrugge and Allan Kielstra and Christopher J.F. Pickett
clump@cs.mcgill.ca, kielstra@ca.ibm.com, cpicke@cs.mcgill.ca

September 2011

Abstract

Thread-level Speculation (TLS) is a technique for automatic parallelization that has shown excellent results in hardware simulation studies. Existing studies, however, typically require a full stack of analyses, hardware components, and performance assumptions in order to demonstrate and measure speedup, limiting the ability to vary fundamental choices and making basic design comparisons difficult. Here we approach the problem analytically, abstracting several variations on a general form of TLS (method-level speculation) and using our abstraction to model the performance of TLS on common coding idioms. Our investigation is based on exhaustive exploration, and we are able to show how optimal performance is strongly limited by program structure and core choices in speculation design, irrespective 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.

1 Introduction

Thread-level speculation (TLS) has been the subject of a large number of research studies, with a wide variety of system proposals and experimental studies [10, 4, 23, 17, 14, 8]. Best performance in such systems, however, typically depends on a complex set of analyses and component assumptions, designed to ensure that TLS resources are focused directly on the most profitable scenarios. This typically requires identifying and avoiding (or efficiently repairing) data-dependencies, reducing the costs of misspeculation, and heuristically and dynamically locating code segments that respond best to a speculative approach.

Despite the many successes, however, generally good performance remains somewhat elusive. Not all benchmarks respond well [8], and the heuristic ability to identify ideal *fork-points* for thread-based speculation tends to be relatively fragile with strong input or program sensitivity. Beyond being complex or having intricate data-dependencies, the precise reasons *why* a program fails to parallelize well are not entirely clear. Part of the problem is in understanding the characteristics of TLS itself; speculation with limited resources is fundamentally a feedback-sensitive technique, where even if decisions are determined to be locally efficient they may nevertheless have important non-local impact by effectively prohibiting future choices. In understanding TLS it is clear that data dependency concerns are extremely important; it is our contention, however, that ignoring the interplay of code structure and speculation model misses a substantial part of the TLS story.

In this work we make an initial foray into understanding how the parallelism found through individual choices implied by specific TLS designs affects global performance. Our approach is to perform a limit study, attempting to determine maximum performance irrespective of and orthogonal to other (still important) concerns of data-dependence. We develop an abstract, flexible and general model of Method-Level Speculation (MLS) as a representative approach to TLS, and perform exhaustive analysis of different behaviours when applied to very basic code idioms. This technique has a number of advantages over existing heuristic performance analyses embedded within specific projects. A methodical approach to analysis clearly exposes the difference between several core MLS designs, and 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 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 clearly reveals several subtle variations in TLS approach.
- We extend our base model with incremental complexity, representing three core forms of TLS, *in-order, out-of-order,* and *nested* threading models. We show how to represent different parent/child signaling disciplines, and can incorporate both the representation of unsafe instructions and multiple forms of TLS overhead.
- We apply our formalism to several basic coding patterns (idioms), experimentally examining the interplay between overall TLS design, fork heuristics, and code structure. We show that even simple programming design differences can result in significantly different parallelization performance, independent of data-dependency considerations.

2 MLS Background

Method-Level Speculation (MLS) is a conceptually straightforward technique for improving performance based on the existence of otherwise idle multiple-CPU resources. At a given method invocation site, an MLS system launches or *forks* a speculative thread to execute the method continuation (i.e., the code following the method call site), while the original parent thread proceeds with normal, non-speculative execution of the method call itself. Upon returning from the method call, the parent thread *joins* with the speculative thread and validates the speculative execution, ideally resulting in parallel execution of the method body and some portion of its continuation.

Other models of TLS direct parallelization to different code structures. *Loop-level* speculation, for instance, performs a very similar activity to MLS but using a loop iteration entry as a fork point—the parent thread forks a speculative thread to execute the subsequent loop iteration at the same time as the parent iteration [6]. *Arbitrary* speculation is also possible, forking speculative threads to execute any given future code sequence [2]. While arbitrary speculation seems most general, all these forms of speculation can in fact subsume each other with appropriate code transformations, conceptual or actual—arbitrary or method-level speculation can be modeled in loop-level as loop bodies branching on the loop index, and loop and method-level speculation can trivially be treated as arbitrary. In this study we used method-level speculation as a general form, making use of the fact that both loop bodies and arbitrary chunks of code can be *outlined* into methods.

There are of course a number of safety and efficiency concerns in any TLS model, and a few specific to MLS as well. Safety is primarily ensured through strong isolation of the speculative execution, and the validation process during joining. Isolation is required to ensure speculative writes do not conflict with non-speculative reads or writes prior to validation. Validation is required to ensure that speculative execution represents behaviour based on the correct input-state of the continuation code, which is potentially affected by the writes of the non-speculative parent. Only once a speculative thread has been appropriately validated should its results be committed to main, non-speculative memory.

Significant performance concerns arise in the implementation of the various required components. Some overhead is of course necessary for thread initialization and for signaling/termination, while isolation implies some cost in modifying code to appropriately buffer speculative writes, and validation requires recording and comparing input assumptions made by the speculative code, perhaps also including thread *abortion* if these assumptions are not correct (misspeculation). For MLS in particular, speculative input-state often includes the output or *return-value* of the method call preceding the continuation, and so the likelihood of successful validation can be improved by return-value prediction, which itself can have non-trivial cost. 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 return-value prediction, and the latter including signaling/termination, validation, and return-value prediction updates. Since many hardware designs allow for read-monitoring at little to no additional cost, we make no attempt to explicitly model that in this work.

Within the scope of assumed, 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 obviously 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 [8]. Importantly, there are strong *feedback* concerns in forking heuristics—CPU resources are limited and technical demands of MLS implementation impose limitations as well, and so 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 input. It is the latter property that we focus on in this work.

3 Modeling MLS

Our system for modeling MLS is based on a simple, stack-oriented program execution model. A program executes sequential code, including properly 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. It is important to note that we do not track data-dependencies or consider misspeculation in this model. Our 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 adds overhead and reduces efficiency, but not does not introduce new possibilities.

Our model thus begins from 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.*

Definition 1 *Let $T = t_1, \ldots, t_n$ be a sequential trace of actions from a properly nested execution, where each of t_i is either a* work *action or a* call. *The* MLS constraint graph *is a directed graph $(V, E = E' \cup C)$, where $V = \{t_1, \ldots, t_n\}$, $E' = \{(t_i, t_{i+1})\}, i = 1, \ldots, n-1$, and C is a set of call-continuation edges, consisting of all (p, q) s.t. p is a call and q is the first statement in the continuation of p. An example is shown in Figure 1.*

The MLS constraint graph works in conjunction with a model of MLS execution. Perhaps the simplest MLS model allows just one speculative thread, and does not permit joining. A speculative child once launched thus "runs to completion." Note that we may also consider this a model of a

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

   A()

b) A → w1 → B → w3 → C → w5 → w4 → w2 → 0
```

c) $$\begin{tikzcd}A \arrow[r] \arrow[rrrrrrrr, bend left, dotted] & w1 \arrow[r] & B \arrow[r] \arrow[rrrrr, bend left, dotted] & w3 \arrow[r] & C \arrow[r] \arrow[rr, bend left, dotted] & w5 \arrow[r] & w4 \arrow[r] & w2 \arrow[r] & 0\end{tikzcd}$$

Figure 1: a) Code, b) an execution sequence given the single call to A(), and c) the corresponding MLS constraint graph; dashed edges are continuation edges.

system where joins involve the parent thread transferring its state to the speculative child, rather than vice versa, as is more typical (and of course not launching further speculation).

The potential behaviour of this model 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.

```
(1) ; (A→w1→B→w3→C→w5→w4→w2) | (0)
(2) A→w1 ; (B→w3→C→w5→w4) | (w2→0)
(3) A→w1→B→w3 ; (C→w5) | (w4→w2→0)
(4) A→w1→B→w3→C→w5→w4→w2→0
```

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 '|'.

Note that we can already observe in this simple execution context that the parallelism generated strongly depends on 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 none of the fork points are selected, and is just sequential execution. Also note that not all sequences represent good fork choices—sequence (1) implies launching a speculative thread that does nothing but terminate. As we will see below it is also possible for a parent thread to immediately join with a just-launched speculative thread. In order to keep our model exhaustive, however, we include even these suboptimal possibilities.

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 "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 definition for a time calculation function τ:

$$\begin{aligned} \tau(t_1, \ldots, t_n) &= \sum_{i=1}^{n} \omega(t_i) && \text{no forking} \\ \tau(S;A|B) &= \tau(S) + F + \max(\tau(A), \tau(B)) + J && \text{forking} \end{aligned}$$

Speedup is of course given by the ratio of the cost of sequential execution to $\tau(T)$.

3.1 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 speculation*, *in-order speculation*, and *nested speculation.*

Perhaps the most straightforward approach is *out-of-order speculation.* In this 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           }
   }                 }

   A()

b) A → B → C → w3 → w2 → w1 → 0
c) ; (A → ; (B → ; (C → w3) | w2) | w1) | 0
```

Figure 3: a) Code, b) a sequential execution sequence, and c) an out-of-order MLS execution assuming an arbitrary number of threads (CPUs) available.

An alternative and symmetric design is to allow speculative children to themselves launch speculative children. This is known as *in-order speculation,* 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 speculation does not, and vice versa. An example of in-order speculation is shown in Figure 4. Note that out-of-order speculation 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.

Finally, one may of course combine out-of-order and in-order techniques, allowing each thread to

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

   A();B();C();

b) A → w1 → B → w2 → C → w3 → 0
c) ; (A → w1) | (; (B → w2) | (; (C → w3) | 0))
```

Figure 4: a) Code, b) sequential execution sequence, and c) an in-order MLS execution assuming an arbitrary number of threads (CPUs) available.

have any number of speculative children, whether the parent thread is speculative or not. This is *nested speculation.* An example is shown in Figure 5.

```
a) A() {              B() {              C() {
     B$_1$()             C$_1$()            workC
     B$_2$()             C$_2$()          }
     workA               workB
   }                   }

   A()

b) A → B$_1$ → C$_{1,1}$ → wC1 → C$_{1,2}$ → wC2 → wB1 →
       B$_2$ → C$_{2,1}$ → wC3 → C$_{2,2}$ → wC4 → wB2 → wA → 0

c) A → ; (B$_1$ → ; (C$_{1,1}$ → wC1) |
                    (; (C$_{1,2}$ → wC2) | wB1)) |
         (; (B$_2$ → ; (C$_{2,1}$ → wC3) |
                       (; (C$_{2,2}$ → wC4) | wB2)) |
            (wA → 0))
```

Figure 5: a) Code, b) sequential execution sequence, and c) a nested MLS execution.

Although it is more difficult to see, nested speculation results in an optimal 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. This is shown in Figure 6.

```
a) A → ; (B$_1$ → ; (C$_{1,1}$ → wC1) |
                    (; (C$_{1,2}$ → wC2) | wB1)) |
         (B$_2$ → C$_{2,1}$ → wC3 → C$_{2,2}$ → wC4 →
               wB2 → wA → 0)

b) A → ; (B$_1$ → C$_{1,1}$ → wC1 → C$_{1,2}$ → wC2 → wB1) |
         (; (B$_2$ → ; (C$_{2,1}$ → wC3) |
                       (; (C$_{2,2}$ → wc4) | wB1)) |
            (wA → 0))
```

Figure 6: a) An out-of-order execution of the trace from Figure 5, and b) an in-order execution of the same trace.

3.2 Signaling, Joining and Stopping

In the above examples the performance of out-of-order and in-order alone can be suboptimal, at least partly because joins are only abstractly present, and our simple run-to-completion model does not allow threads to be reused after joining—in the case of pure out-of-order threading, launching a speculative thread forever prevents any further speculation within the continuation code, even if the parent execution was relatively short. A practical design, however, allows for thread reuse, redeploying the speculative thread resource once a speculation 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. There are in fact two main approaches to thread joining that are possible and can be represented in our system: *forward-signaling* and *backward-signaling*. We address both below, as well as how we could incorporate the existence of speculative *unsafe* instructions, which prevent speculation from proceeding further.

Forward-signaling is perhaps the most common form of parent/child join-synchronization used in MLS. The main idea is that 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 speculative child-thread in subsequent execution and thus further speculation within the continuation of the first speculative execution's original scope. The end effect is similar to allowing a nested threading model, but differs by limiting the choice of fork points in the continuation to those encountered after the join point with the speculative child. An example trace and resulting out-of-order MLS execution, with and without forward-signaling is shown in Figure 7. Note how the MLS execution without signaling has limited parallelism, greatly improved by the use of forward-signaling.

```
a) W() { work }

   W() W() W() W()

b) W → work → W → work → W → work → W → work → 0
c) (no signal) ; (W → work) |
                 (W → work → W → work → W → work → 0)
d) (w/ signal) ; (W → work) | (W → work) [join]
               ; (W → work) | (W → work)
```

Figure 7: a) Code, b) Trace of a nested execution, c) out-of-order MLS execution without explicit signaling, d) out-of-order MLS execution with forward-signaling. Note that we assume equal time (cost) to execute each work unit.

Forward-signaling applies most naturally to out-of-order execution, but unfortunately is not as effective for in-order execution. With in-order speculation, 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 and reuse 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 execution resource to launch a subsequent speculative child prior to joining. An example of in-order execution with and without backward signaling is shown in Figure 8. As with forward-signaling, backward-signaling produces an effect similar to nested speculation; in this case differing in that parent threads may not launch more speculation children until a speculative child has terminated.

```
a) A() {              B() {
     B()                workB
     workA              }
   }

   A()

b) A → B → workB → workA → 0
c) (no signal) ; (A → B → workB → workA) | 0
d) (w/ signal) ; (A → ; (B → workB) | workA) | 0
```

Figure 8: a) Code, b) Trace of a nested execution, c) in-order MLS execution without signaling, d) in-order MLS execution with backward-signaling.

Backward-signaling has the disadvantage that terminated speculative thread states need to be retained until parent execution reaches the corresponding continuations. In the rest of this work we model only forward-signaling. Explicit modeling of backward-signaling is possible in our design, but adds significant further complexity and so is left for future work.

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 the 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 function call, A is the function body, B is the continuation up to the point at which the speculative thread is joined, and C is the remaining execution, giving us an overview equation:

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

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 terminated, with the process identical to stopping due to a parent signal. In the case of in-order or nested speculation, attempts to execute the unsafe instruction will result in further recursive unwindings of speculative parents, eventually reaching the main, non-speculative parent.

Exhaustive Algorithm Figure 9 formalizes the notions discussed in this section. Given a sequential execution, it expresses all possible in-order, out-of-order, or nested MLS executions, accommodating forward-signaling, non-speculative instructions, and limited thread resources.

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

```
Let $T = t_1, t_2, \ldots, t_n$ be a sequential trace of actions.
MLS($T$,$\sigma$,time) =
  for all $S$ = preamble($T, \sigma$) s.t. $\tau(S) \leq$time
    let ($t_{|S|+1}$,$t_b$) be a continuation edge
    $T_A = t_{|S|+1}, \ldots, t_{b-1}$
    for all $\sigma_1, \sigma_2$ = $\sigma$-1,0         // for out-of-order
                           0,$\sigma$-1         // for in-order
                           split($\sigma$-1)  // for nested
      for all $A$ = MLS($T_A$,$\sigma_1$,time-$\tau(S)$-$F$)
        $T_B = t_b, \ldots, t_n$
        for all $B$ = MLS($T_B$,$\sigma_2$,$\tau(A)$)
          $T_C = t_{|S|+|A|+|B|+1}, \ldots, t_n$
          $\tau(S; A|B) = \tau(S) + F + \max(\tau(A), \tau(B)) + J$
          for all $C$ = MLS($T_C$,$\sigma$,time-$\tau(S; A|B)$)
            $\tau(T) = \tau(S; A|B) + \tau(C)$
            return $S$ ; $A$ | $B + C$
```

Figure 9: Algorithm for enumerating in-order, out-of-order, or nested 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 function 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.

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 function call (if not then then the result is just a single, sequential execution of T). The function call defines the split between the preamble S, the function 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 nested 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, after A and B join. 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 prior, recursive execution). For assessing speedup, the total time taken is calculated and associated with the input sequence.

Note how the timeouts are used to enforce forward-signaling (only). To model backward-signaling we would need to invert the dependencies of 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. Modeling both forward *and* backward-signaling, *i.e.,* a complete *bidirectional-signaling* model, could

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 [19].
tail	Tail-recursion, 10 levels deep, each call executing a work function before the recursive call [19].
treeAdd	A double head-recursion, corresponding to a recursive descent of a binary tree down to 3 levels (7 units of work total). Modeled after the “TreeAdd” JOlden benchmark.

Figure 10: Synthetic benchmark suite. Note that these represent control-flow abstractions only, and do not include data-dependencies.

be computed as a fixed-point, balancing the time consumed by A given the resources passed back from B, the amount of which recursively depends on the time consumed by A. Given the greater importance of forward-signaling, and the complexity of modeling bidirectional-signaling, we here restrict ourselves to just forward-signaling.

4 Experimental Analysis

An experimental investigation is performed by executing the algorithm of Figure 9 under different parameter assumptions and applied to different program structures, represented through sequential program traces. Although these are synthetic traces, not including data dependencies or misspeculation behaviours, they show how choices of in-order, out-of-order, or nested speculation affect potential performance, the impact of different overhead costs (fork and join), and how the resulting parallelism is altered by trivial variation in code structure.

As a benchmark suite we use small program traces based on coding-idioms that correspond to common programming styles. These are summarized in Figure 10 and partly extend the basic models considered in work by Pickett *et al.* [19]. For most of these idealized examples 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 [21], and the work-weight is chosen to be much larger in proportion. Section 4.2 further investigates how the relative weight of work and overhead affects performance, providing justification for our use of such simplistic choices.

4.1 Speedup

A central question in our analysis is how the combination of code structure and MLS design relate to potential 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 nested), under forward-signaling for all of our benchmarks. We consider 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, 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.

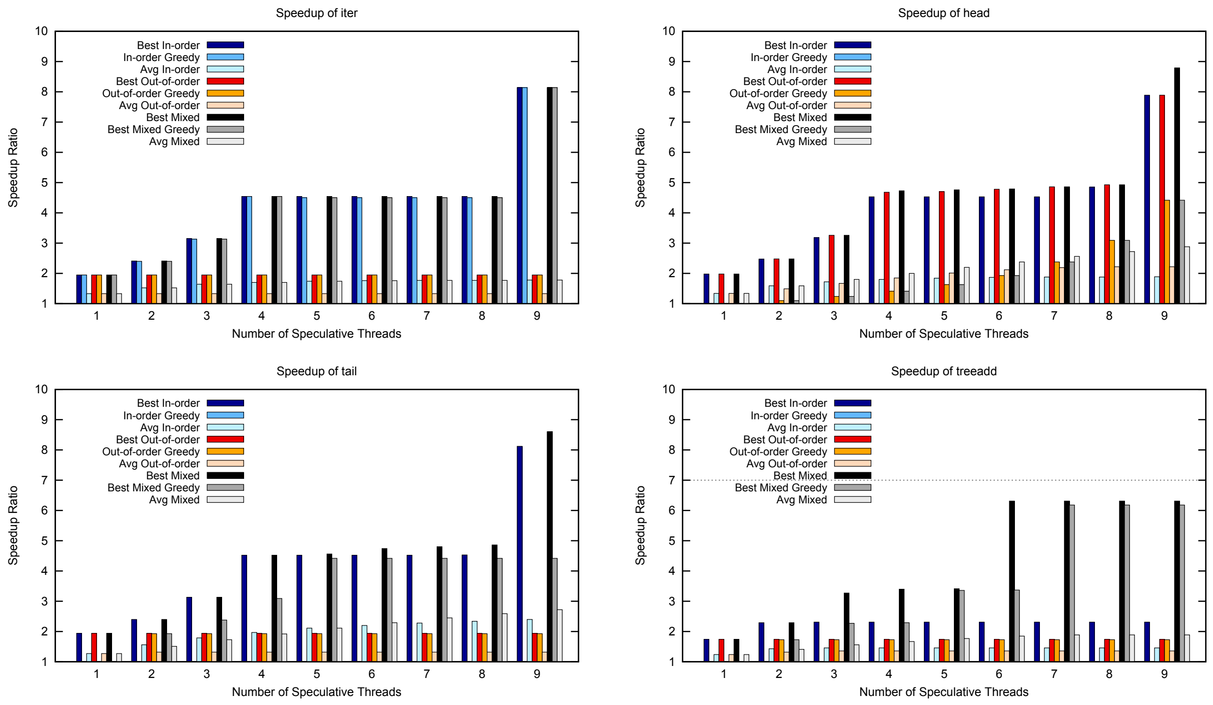

Figure 11: Speedup for benchmarks given different maximal thread resources, thread models, and fork heuristics. A graph is shown for each benchmark; for each number of available speculative threads, maximal speedup, greedy speedup, and average speedup are grouped and shown for in-order, then out-of-order, and finally mixed speculative strategies respectively. Maximum theoretical speedup is 10 in all cases except for treeAdd, where to maintain symmetry of the tree-based recursion maximum theoretical speedup of 7 is used, indicated by the dotted horizontal line.

Note that with pure in-order and out-of-order MLS only a single greedy choice is possible in each case; with a nested model, however, depending on how thread resources are divided between the in-order and out-of-order strategies (a choice made at each method call) there are multiple possible greedy results. Results for the greedy option under nesting 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 11. 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 nested 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 treeAdd, where the pure strategies are basically limited to exploiting one branch down the tree while a nested approach lets the best individual strategy be selected at each branch in the descent.

Average performance is mainly interesting in providing evidence of the extent of bias toward sub-optimal 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 treeAdd greedy works well for out-of-order and of course nested, 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. This may be partly considered an artifact of limitations in our MLS abstraction, and a model where idle threads may be repurposed after completion but before joining (*i.e.,* backward-signaling), would improve the results for in-order.

4.2 Weight Sensitivity

An important and interesting question for TLS is to what extent overhead has an impact on potential performance. Most TLS studies approach speculative designs assuming forking, joining and other overheads will be a major bottlenecks for performance. Certainly in software these costs can be relatively high [17, 18]. Hardware and hybrid designs thus often expend significant effort on features that mitigate these costs, in some cases reducing them to just around the same order of magnitude as normal instructions, at least in simulation; *Mitosis,* for example, assumes just 5 cycles for a thread fork and 20 for a join [21], and we have replicated those assumptions in our study here. In this experiment we thus vary fork and join costs from 0 to 10000 work units, to determine how sensitive our behavioural observations are to the actual overhead assumptions. Speedup results are shown in Figure 12 with respect to fork cost; for simplicity, in this data joins are assumed to be twice as expensive as the forks.

In an overall sense, a degradation in scalability, roughly proportional to the increase in overhead is to be expected, and indeed this is evident in the experimental data. The effect is not entirely uniform, however, with head, tail, and treeAdd showing more resilience (degrading less quickly) than iter, at least until overhead costs become very large. The extent of performance change can in fact be more closely related to the maximum speedup the benchmarks exhibit—in the absence of misspeculation, overhead in our abstraction is entirely created at thread forks and joins, and so the more threads used (and thus greater potential speedup) the more overhead sensitivity. It does, however, show that overhead is a largely separate and independent concern from code structure and one that does not overly perturb our results—relative differences do not strongly depend on subtle changes to overhead values, as long as overhead does not overwhelm thread-length, a factor already well-known to be important in fork heuristics.

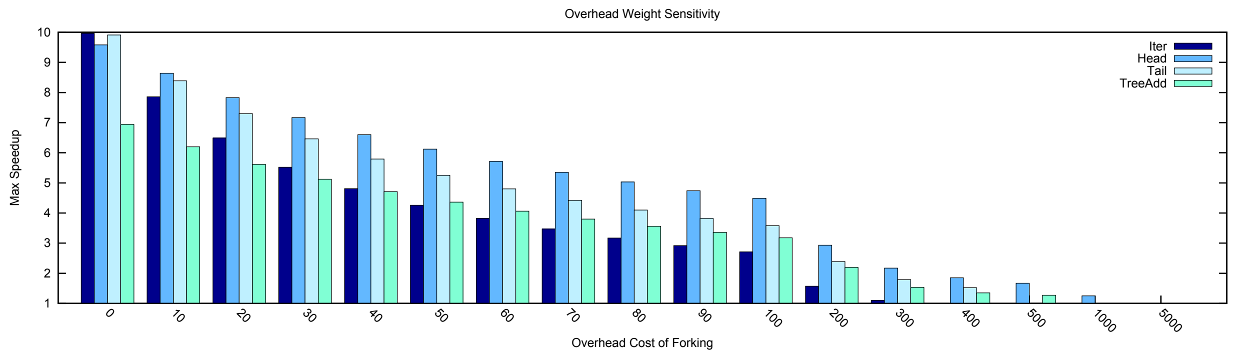

Figure 12: Maximum speedup under different overhead assumptions (joining is twice the cost of forking), given nested threading and forward-signaling, and a maximum of 9 speculative threads. Note that the X-axis scale is only piece-wise linear.

4.3 Code Structure

Our basic thesis is that the interplay of code structure and MLS design has a large impact on speedup. Certainly, this is true of core, algorithmic structure, as shown in Figure 11; it is also true, however of quite trivial code modifications. Here we consider three, conceptually simple changes to each of our benchmarks made by adding a small amount of work and a method call in three symmetric positions:

prefix	The benchmark has a method call prepended to it, such that the main benchmark code is executed in the continuation of a short (100 units) method call.
wrap	A call is made to the main benchmark code, with a short (100 units) amount of work performed afterward.
suffix	A short (100 units) amount of work is done in a call which is performed after (in the continuation of) the main benchmark code.

In terms of optimal performance, these changes are very minor—the addition of a small amount of work and an extra call can be accommodated by allocation of an extra thread resource, reducing threading efficiency, but otherwise having very minimal impact on maximal possible speedup. The impact on more realistic fork heuristics is however quite large, and Figure 13 shows how speedup is affected by these code changes for the greedy strategy.

First, as mentioned for optimal threading, the extra method call used results in slightly less efficient use of threads, an effect most obvious in the case nested MLS. Most striking, however, is how radically performance is altered for the different benchmarks in the case of the less-comprehensive, pure in-order and out-of-order models. Iter, for instance, has prefix and suffix versions similar to the normal version, showing slight reductions owing to the small amount of serial work introduced before or after the main code. Wrapping, however, has a large impact for in-order threading, effectively eliminating all speedup. In the wrapped version in-order threads are captured in the small amount of work that constitutes the continuation of the wrapped call. These threads must then wait for the main thread to join before releasing resources; again, demonstrating the value of being able to archive thread state and reuse unjoined threads for best performance.

Other benchmarks show some initially surprising and counter-intuitive behaviours for the in-order approach. The greedy strategy was originally ineffective in pure in-order for head, tail, and treeAdd,

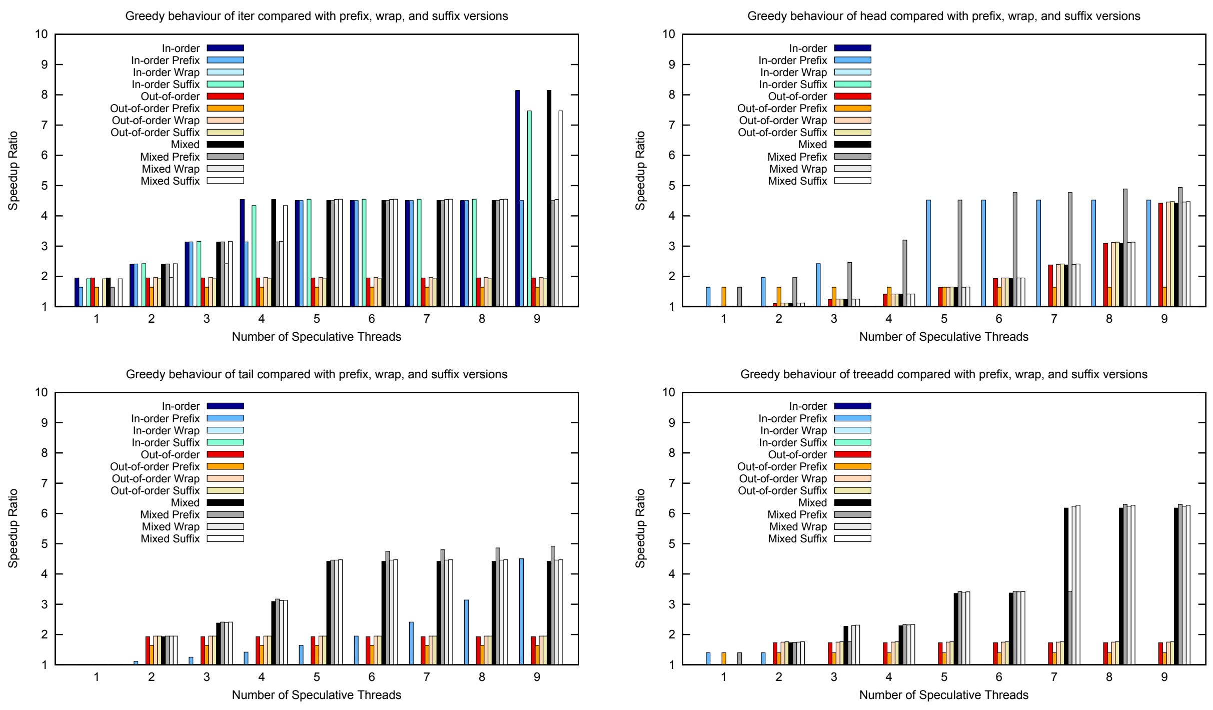

Figure 13: Speedup values under a greedy strategy for benchmarks given different maximal thread resources. For each number of available speculative threads, speedup is shown for normal (i.e., the same greedy data from Figure 11), prefixed, wrapped, and suffixed versions of the benchmarks grouped together, for in-order, then out-of-order, and finally nested MLS.

but here shows great improvements when new work is introduced in the prefix variation, especially for head. Improving speedup by adding work illustrates the complex, non-linear feedback properties found in speculative thread allocation. Here the effect is mainly due to the way prefixing moves the main benchmark code into a continuation, allowing in-order (and consequently nested) MLS to be better exploited. It is interesting that the poor behaviour associated with being unable to repurpose in-order threads is largely eliminated by this simple code transformation, contradicting the need for unjoined thread-state storage mechanisms found in the case of wrapping and iter.

Out-of-order turns out to be more resilient to wrapping and suffixing, showing little impact in all benchmarks from these modifications. Prefixing, however, has a significant but in this case negative impact, most clearly evident on the head benchmark. In this situation the good scaling of out-of-order speedup on the normal version is absent when prefixing is applied. This occurs because of the way the prefix code causes better opportunities to be missed. In the normal greedy version, threads are launched as the head recursion descends, and each is given ample time to complete its computation while the main thread descends. In the prefixed version the main benchmark body in the prefix continuation is initially executed by a single speculative thread due to the out-of-order model, and this thread has sufficient time to descend several levels before the main thread joins it, thread resources are recovered and the system can begin again spawning multiple threads as it completes the descent.

5 Related Work

A wide variety of TLS [10] (and MLS [4]) 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 [23], 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 [17]; 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 which uses the virtual memory system to isolate "possibly-parallel" regions [7]. 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.

Whatever the threading model, determining where and when to fork threads is one of the fundamental challenges of a TLS system. As well as the basic safety problem of avoiding or repairing data-dependencies, in order 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 proportion of unnecessary overhead caused by lack of actual parallelism [26]. 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 [27]. The recent *POSH* system uses several optimizations as part of fork (task refinement) heuristics, considering thread-length, dependency and profiler information [14]. 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 as others have reported with optimized out-of-order designs [22].

Abstract models of parallel execution have been of academic 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 [12]. 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 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 [16]. 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 [14].

A more detailed and hardware-specific abstraction of TLS has been used to build a complete TLS taxonomy. Garzarán *et al.* focus on buffer-management, how or whether distinct (speculative) versions of data are differentiated and merged in a TLS system, and define a taxonomy that describes approaches in those terms [11]. This relates to our work in that low-level considerations of the number of data versions, access costs, and ordering constraints imply different speculative control strategies—their *SingleT* category, for example, maps onto a system where a parent thread can have only one speculative child, while *MultiT/SV* allows for a limited form of multiple-child speculation, as long as a specific datum is not duplicated more than once. They conclude that supporting multiple data versions is an overall effective means of improving performance; this does not map precisely to either out-of-order or in-order designs, but is instead a general requirement on any system that allows more than one speculative child.

The abstraction we investigate here does not try to form an explicit taxonomy, but instead builds on recent results described by Pickett *et al.* on modeling and understanding MLS [19]. Their work does not perform exhaustive analysis, aiming more at a visualization system, but has inspired our basic approach, and we have used several of the code idioms they describe 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 [3]. This is applied during runtime to help identify loops appropriate for TLS execution, allowing the rest of the *Jrpm* hardware-software hybrid system [5] 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 [9]. 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 [25].

Dou and Cintra take a more exhaustive 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 [8]. To maintain practicality within a compiler framework, they do not extend their model to nested loops or recursion. Interestingly, even with this intricate model simulation results show a broad range of speedup and slowdowns depending on the benchmark.

Quantitative approaches have also been extended to the similar world of transactional memory designs. von Praun *et al.*, for example, define an approach using both control and data-oriented density metrics, heuristically measuring the extend of dependency between potentially parallel critical sections, and thus their suitability with respect to different parallelization modes [24]. They demonstrate their technique on several explicitly parallel algorithms, although the model would naturally extend to TLS.

Given the complexity and variable results from TLS designs, considering completely different parallelization paradigms is attractive, and other, highly general approaches to automatic parallelism are being aggressively explored. The *Galois project* [20] focuses on data-parallelism of irregular applications, where it can be demonstrated that a significant amount of potential parallelism exists in terms of data locality, despite the complexity of extracting it through standard control flow [13]. A data-parallel approach is also at the basis of Lublinerman *et al.*'s recently proposed *Chorus* programming model [15]. Chorus exposes dynamic data-partitioning to the programmer through "object assemblies," providing convenient mechanisms to define, split, and merge localized data. Our results here echo the motivations for these approaches, control flow is indeed a central concern with respect to efficient parallelization, although as we show success and failure in parallelization can be at least partially back-attributed to the combination of specific code structures and the basic approach to parallelism used, and so may be subject to greater control.

6 Conclusions and Future Work

Our work here complements 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 of even small code idioms we demonstrate the large impact code structure has on potential speedup, and show how structure and fork choices can interact to drastically alter performance. Better understanding of this behaviour is a basis for developing code modifications and tailoring fork strategies to maximize performance, in a way largely orthogonal to data dependency considerations.

There are many interesting aspects of speculation we can further explore with our model. Adding in misspeculation costs is trivial, although a full consideration would greatly magnify the effort required to perform exhaustive consideration. A more useful extension may be to incorporate backward-signaling, for a more complete model of complex (state-saving) MLS designs. Our main interest, however, is in scaling up the design as much as possible, and using the results as a further source of heuristic information to guide threading decisions within a complete MLS prototype.

Acknowledgements

This research was supported by the IBM Toronto 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: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 162–173, New York, NY, USA, 2000. ACM.

[2] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '02)*, pages 99–108. ACM Press, Aug. 2002.

[3] M. Chen and K. Olukotun. TEST: a tracer for extracting speculative threads. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 301–312, Washington, DC, USA, 2003. IEEE Computer Society.

[4] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *PACT'98: Proceedings of the 7th International Conference on Parallel Architectures and Compilation Techniques*, pages 176–184, Oct. 1998.

[5] M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *ISCA'03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 434–446, June 2003.

[6] M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03)*, pages 13–24, June 2003.

[7] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 223–234. ACM, 2007.

[8] J. Dou and M. Cintra. A compiler cost model for speculative parallelization. *ACM Trans. Archit. Code Optim.*, 4, June 2007.

[9] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 71–81. ACM, 2004.

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

[11] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(3):247–279, Sept. 2005.

[12] J. Greiner and G. E. Blelloch. A provably time-efficient parallel implementation of full speculation. *TOPLAS: ACM Transactions on Programming Languages and Systems*, 21(2):240–285, 1999.

[13] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 3–14, New York, NY, USA, 2009. ACM.

[14] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: a TLS compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 158–167, New York, NY, USA, 2006. ACM.

[15] M. Méndez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali. Structure-driven optimizations for amorphous data-parallel programs. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, pages 3–14, New York, NY, USA, 2010. ACM.

[16] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *PACT'99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 303–313, Oct. 1999.

[17] C. J. F. Pickett and C. Verbrugge. SableSpMT: A software framework for analysing speculative multithreading in Java. In *PASTE'05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 59–66, Sept. 2005.

[18] C. J. F. Pickett, C. Verbrugge, and A. Kielstra. Adaptive software return value prediction. Technical Report SABLE-TR-2010-3, Sable Research Group, School of Computer Science, McGill University, Montréal, Québec, Canada, Apr. 2010.

[19] C. J. F. Pickett, C. Verbrugge, and A. Kielstra. Understanding method level speculation. Technical Report SABLE-TR-2010-2, Sable Research Group, School of Computer Science, McGill University, Montréal, Québec, Canada, Apr. 2010.

[20] K. Pingali. Towards a science of parallel programming. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 3–4. ACM, 2010.

[21] 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 *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 269–279, New York, NY, USA, 2005. ACM.

[22] 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: Proceedings of the 19th Annual International Conference on Supercomputing*, pages 179–188, June 2005.

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

[24] C. von Praun, R. Bordawekar, and C. Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 185–196, New York, NY, USA, 2008. ACM.

[25] S. Wang, X. Dai, K. Yellajyosula, A. Zhai, and P.-C. Yew. Loop selection for thread-level speculation. In *LCPC'05: Languages and Compilers for Parallel Computing*, volume 4339 of *Lecture Notes in Computer Science*, pages 289–303. Springer, 2006.

[26] F. Warg and P. Stenström. Improving speculative thread-level parallelism through module run-length prediction. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 12.2–, Washington, DC, USA, 2003. IEEE Computer Society.

[27] J. Whaley and C. Kozyrakis. Heuristics for profile-driven method-level speculative parallelization. In *ICPP'05: Proceedings of the 2005 International Conference on Parallel Processing*, pages 147–156, June 2005.