McGill University
School of Computer Science
Sable Research Group

Memory Abstractions for Speculative Multithreading

Sable Technical Report No. 2008-3

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

September 25th, 2008

Abstract

Speculative multithreading is a promising technique for automatic parallelization. However, our experience with a software implementation indicates that there is significant overhead involved in managing the heap memory internal to the speculative system and significant complexity in correctly managing the call stack memory used by speculative tasks. We first describe a specialized heap management system that allows the data structures necessary to support speculation to be quickly recycled. We take advantage of constraints imposed by our speculation model to reduce the complexity of this otherwise general producer/consumer memory problem. We next provide a call stack abstraction that allows the local variable requirements of in-order and out-of-order nested method level speculation to be expressed and hence implemented in a relatively simple fashion. We believe that heap and stack memory management issues are important to efficient speculative system design. We also believe that abstractions such as ours help provide a framework for understanding the requirements of different speculation models.

1 Introduction

Speculative multithreading (SpMT), also known as *thread level speculation* (TLS), is a dynamic parallelization technique that depends on out-of-order execution, buffering, and rollback to achieve speedup of single-threaded programs on multiprocessors. SpMT works by splitting off multiple speculative child tasks from a single non-speculative parent thread, each executing on its own processor, and joining them at a later time. Long studied in hardware, SpMT has recently shown promise in software designs [7, 24]. Our work in particular has focused on *method level speculation* for Java, wherein tasks are created and joined on method invocation and return, and wherein call stacks are always well-behaved.

Software SpMT implementations have specific memory management requirements with respect to both heap memory for internal data structures as well as call stack memory used for local variables within the speculative tasks themselves. In the former case the main concern is efficiency: support for speculation typically requires structures that grow in size, cross both processor and non-speculative/speculative runtime code boundaries, and moreover get frequently allocated and discarded. Recycling such structures efficiently for SpMT is non-trivial not only because of the high frequency of allocation, but also because the memory migration between threads and processors creates a multithreaded and multiprocessr producer/consumer problem. Stack data adds further complexity: in order to propagate local variables to and from speculative tasks, stack frames must be copied between thread-specific call stacks. Although seemingly straightforward, a flexible design that allows for relatively arbitrary speculative task creation can be difficult to describe and implement. A clear expression of the requirements of different speculation models is important for developing efficient and correct communication models for local data, and also for understanding the trade-offs between the improved potential parallelism and memory costs of different speculation models.

We present two designs: one for the efficient management of internal heap structures, and another for satisfying stack copying requirements in a flexible manner. The first design takes advantage of the constraints imposed by our speculation system. In particular, rather than develop a general multithreaded memory manager, we assume an ownership model of data and use that to recycle the complex speculative child task structures used in a typical SpMT system. The second design is more complex: it ensures that accesses to local variables are propagated between non-speculative parent and speculative child tasks safely and efficiently, and it accomodates the requirements of different method level speculation models. We provide an abstraction using a simple notation based on call stacks, and using this stack model we specify how local variable data must be moved for two kinds of speculative task creation: *in-order nesting*, wherein a specu-

lative task can create a doubly-speculative task its own, and *out-of-order nesting*, wherein a single thread or task creates multiple speculative tasks of its own, one per stack frame.

1.1 Contributions

- A simple solution for constrained producer/consumer memory management problems on multiprocessor systems that recycles complex, aggregate data structures.
- A multithreaded call stack abstraction for method level speculation that supports both in-order and out-of-order speculation. We show that by using simple operations on call stacks, various models of method level speculation can be described, understood, and implemented using a common framework.

2 A Simple Custom Ownership-Based Multithreaded Allocator

Our initial SpMT implementation allowed parent threads to allocate multiple child tasks, but not for child tasks to create doubly-speculative child tasks of their own. Child tasks were enqueued on a concurrent priority queue, and after execution by a helper/worker thread their parent would abort or commit them, freeing their memory. Since each parent eventually freed all of its children, all system resources were reclaimed and there was no malloc/free producer/consumer problem.

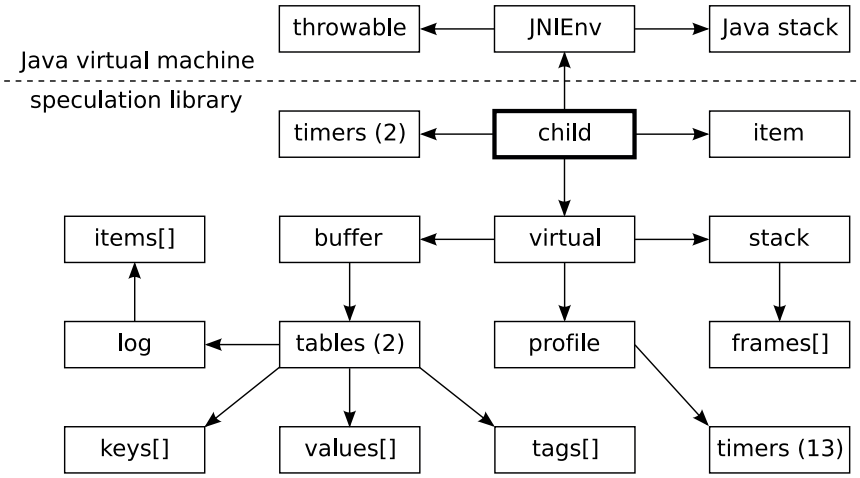

Figure 1: Runtime child data structure.

However, we were initially creating a new instance of the runtime child structure shown in Figure 1 for each new child task. Since our speculation model allows for a high frequency of task creation, up to once for every non-speculative method invocation, and since each child structure has 37 separately allocated subcomponents, this quickly led to a performance bottleneck with the otherwise suitable Lea allocator [16].

We realized that each child structure is an *ownership dominator tree* [19] rooted by the `child` sub-object, such that all other sub-objects are only reachable through it, at least at allocation and deallocation time. Since all child structure instances have the same shape, the introduction of a new child 'reset' operation on allocation allowed us to give each parent thread a child structure freelist and avoid excessive calls to malloc and free.

Later profiling for overhead [23] revealed that it would increase the exposed parallelism and thus benefit speculation performance to allow a child task to create new child tasks of its own. Under our speculation model, only a non-speculative parent thread can commit child tasks, and consequently it is the original parent thread that ultimately frees the combined resources for both child generations. This creates a multithreaded and multiprocessor producer/consumer allocation pattern, because although allocation occurs in either parent threads or helper/worker threads, deallocation only occurs in parent threads. If we now used our simple per-parent thread freelists, child structure memory would pool up and not get reused. On the other hand, if we used a drop-in multithreaded malloc replacement such as Hoard [2], we would need to significantly alter our SpMT implementation so as not to call malloc and free 37 times more than required.

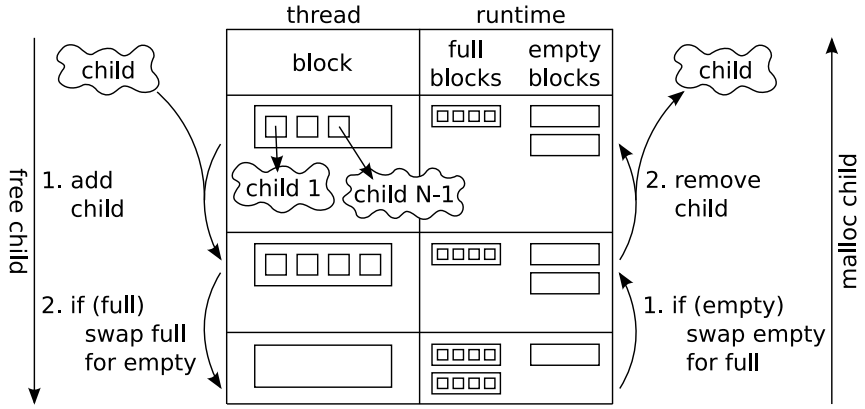

Figure 2: High level illustration of child malloc and free.

Our solution is to use a custom multithreaded memory allocator, as depicted in Figure 2. On the left a child is freed to a local parent thread block of child pointers. If that block becomes full it is exchanged for an empty one via global synchronization at the runtime level. The malloc process is exactly the inverse, exchanging an empty block for a full one if necessary and then producing a child for the current parent or helper/worker thread. Larger block sizes reduce the need for global synchronization, albeit at the expense of extra memory consumption. Figure 3 provides an implementation where the only actual calls to malloc are inside `set_create` and `child_create` on lines 41 and 43.

This scheme has the following advantages: 1) functionality across a library–VM interface: our library calls back into a JVM to create an appropriate thread context; 2) support for child sub-structure type opacity; 3) minimal initialization costs; 4) implementation simplicity; 5) support for dynamically resizable sub-structures, here the buffer and stacks; 6) portability; 7) no external dependences; 8) no synchronization operations in the common case, namely allocating or freeing a child task from or to a local thread block; 9) memory consumption proportional to the maximum number of live tasks.

It also has disadvantages: 1) potential lack of locality between child sub-structures; 2) lack of locality between processors: an individual child task may visit 3 different cores, the allocating, executing, and freeing ones; 3) no reclamation of excess child task memory; 4) lock-based synchronization in the uncommon case, namely exchanging empty and full blocks between a thread and the global pool of blocks; 5) lack of automation and general purpose applicability.

We believe that this particular producer/consumer pattern is important for flexible software SpMT, and hence refinements to this scheme or completely different solutions are worth exploring. We also believe that high-level memory management of ownership dominator trees may be a generally useful paradigm.

```
1  child_t * malloc_child (thread_t *thread) {
2    if (is_empty (thread->block))
3      thread->block = swap_empty_for_full
4        (thread->runtime, thread, thread->block);
5    return remove_child (thread->block);
6  }
7
8  void free_child (thread_t *thread,
9                   child_t *child) {
10   add_child (thread->block, child);
11   if (is_full (thread->block))
12     thread->block = swap_full_for_empty
13       (thread->runtime, thread, thread->block);
14 }
15
16 set_t * swap_empty_for_full (runtime_t *runtime,
17                              thread_t *thread,
18                              set_t *empty) {
19   set_t *full;
20   acquire (runtime->blockset_lock, thread);
21   add_block (runtime->empty_blocks, empty);
22   full = (is_empty (runtime->full_blocks)) ?
23     block_create () :
24     remove_block (runtime->full_blocks);
25   release (runtime->blockset_lock, thread);
26   return full;
27 }
28
29 set_t * swap_full_for_empty (runtime_t *runtime,
30                              thread_t *thread,
31                              set_t *full) {
32   set_t *empty;
33   acquire (runtime->blockset_lock, thread);
34   add_block (runtime->full_blocks, full);
35   empty = remove_block (runtime->empty_blocks);
36   release (runtime->blockset_lock, thread);
37   return empty;
38 }
39
40 set_t * block_create (void) {
41   set_t *block = set_create ();
42   while (!is_full (block))
43     add_child (block, child_create ());
44   return block;
45 }
```

Figure 3: Source code for child malloc and free.

3 A Stack Abstraction for Method Level Speculation

The method level speculation model is based on the creation of a child speculative task at method invocation. This child executes the 'continuation' code that follows the invocation, and gets joined when the creating non-speculative parent returns from its call. The child executes in a *strongly isolated* fashion, such that it is effectively sandboxed until the parent joins, validates, and decides to commit it. This model depends upon return value predictors as well as dependence buffering of heap/static reads and writes via some form of transactional memory [24].

Assuming we are operating at the method granularity in a Java virtual machine, we can exploit Java's well-defined call stack behaviour for child task creation. In particular, a child task can buffer an entire stack frame at once from its parent, such that speculative accesses to local variables do not need further buffering. Although there exists significant prior work on method level speculation, there is no in-depth exploration of legal operations at the stack frame level of abstraction, which we now seek to provide.

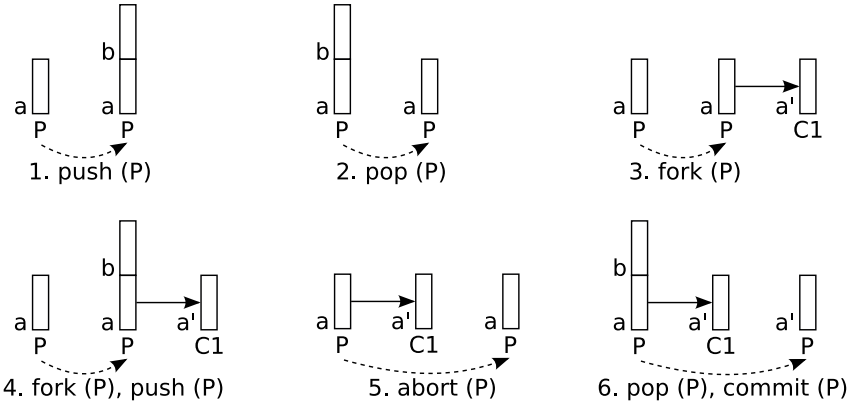

Figure 4: Parent thread operations.

Figure 4 illustrates the actions a non-speculative parent thread **P** can take. In 4.1 and 4.2, **P** pushes and pops stack frame **b** starting from **a**; note that unique frames are assigned unique names. These are the expected stack operations from a sequential execution context. In 4.3 **P** forks a new child **C1**, and hence **C1** gets **a'**, a private version of **P**'s current stack frame **a**. Barring exceptions, method invocations follow forks as shown in 4.4. In 4.5 **P** aborts **C1** by simply deleting it. Aborts may occur for 4 reasons: 1) **P** returns to the fork point and finds a misspeculation in **C1**; 2) references on **P**'s stack are altered by the garbage collector, in which case the entire stack is scanned looking for children to abort; 3) an exception is thrown out of the frame **P** pushed after forking; 4) an exception is thrown while **P** tries to push a new frame. Aborts must occur before executing an exception handler in case the handler attempts to fork a child. Finally in 4.6 **P** commits **C1**, copying **a'** over **a**. Commits are always preceded by a return from a successful method invocation.

Figure 5 illustrates the general actions a speculative child task **C1** can take. There are special restrictions on these actions shown in Figures 6–8. In 5.1 **C1** pushes **c**, distinct from **b** and following it in sequential execution order. In 5.2 **c** is popped; **C1** is free to pop any frame that it has pushed. Upon returning to a frame pre-existing **C1**, that frame must be copied from **P** as shown in 5.3. This lazy copying allows for relatively arbitrary speculation and is a critical performance optimization. 5.4 shows **C1** which copied **a** to **a'** and subsequently pushed **d** and **e**. Upon commit, **P** copies over the entire range of live stack frames from **C1** and resumes at the point where it stopped in **e**. If **P** were to commit **C1** before **d** and **e** were pushed the end result would be **a'** alone for **P**. As a safety constraint **C1** can never invoke nor return to a native or synchronized method, although **P** is always free to.

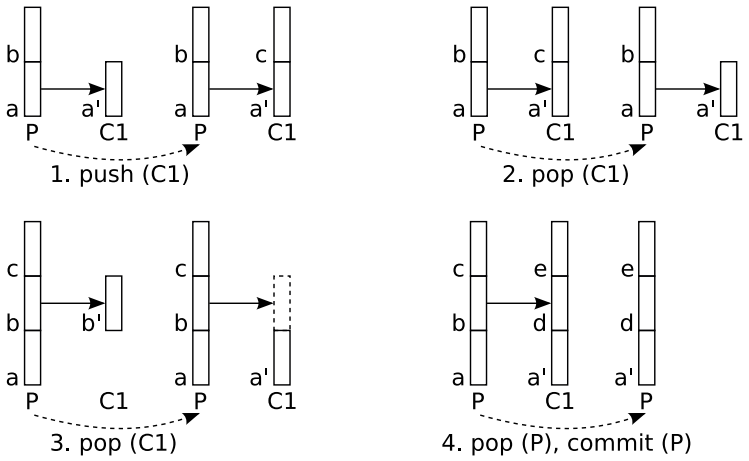

Figure 5: Child task operations.

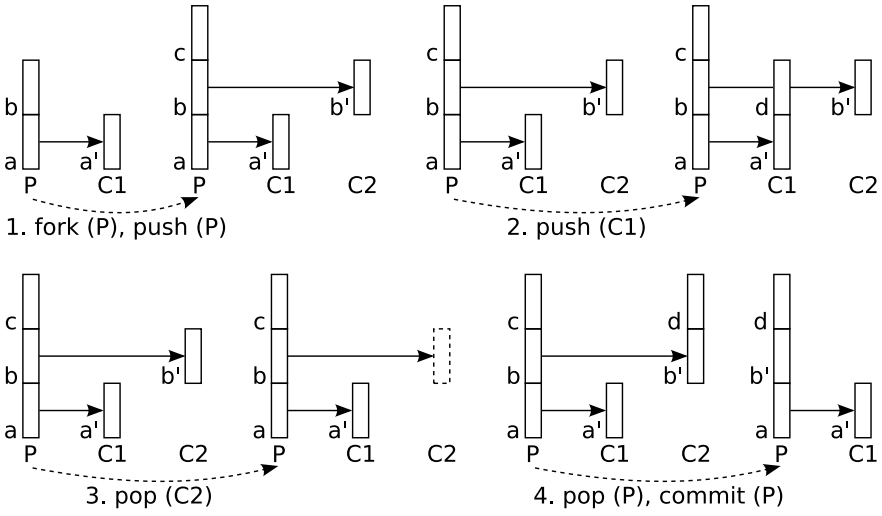

Figure 6: Out-of-order nesting: many children per parent.

The speculation model developed thus far only allows for one child per parent thread. Figure 6 extends it to allow **P** to create one child per non-leaf stack frame, a kind of *out-of-order nested speculation*. In 6.1 **P** already has **C1** in **a** when it creates **C2** in **b**. Naturally, this does not inhibit the progress of **C1**, which pushes **d** at the same height as **b** in 6.2. However, **C2** cannot return to a frame in which **P** already has a child; in 6.3 **C1** is an *elder sibling* of **C2**, and **C2** simply pops **b'** and stops execution. If **P** were then to commit **C2** it would end up in **a** without copying **b'**. An implementation might support merging of **C1** and **C2**, but a dynamic profiling system can instead record encounters with elder siblings and input this data to fork heuristics to avoid creating children likely to collide. Finally, in 6.4 **P** commits **C2**, getting frames **b'** and **d**, and this has no impact on **C1**. In general, for two frames **x** and **y** where **y** has been pushed after **x**, everything reachable from **y** happens before anything subsequently reachable from **x**.

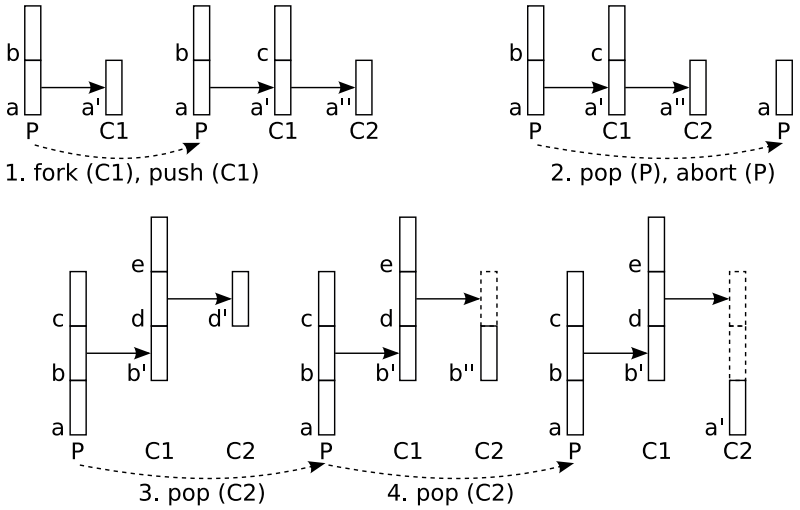

Figure 7: In-order nesting: children of children.

Out-of-order speculation provides extra parallelism, but the *in-order nested speculation* depicted in Figure 7 provides even more. In 7.1, child **C1** creates **C2** and enters **c**. Here **C2** gets **a"**, its own local version of **C1**'s **a'**. If **C1** pops **c** then it must stop execution; compare with 6.3, the other case where speculative tasks collide. 7.1 illustrates how method level speculation can subsume loop level speculation if a loop body is extracted into a method call [5]: **a** could execute the start of the loop, fork **C1** and immediately enter **b** to execute the extracted body. **C1** would then fork **C2** and immediately enter **c** to execute the second iteration of the extracted body. The recursive structure of in-order nesting requires that aborts be recursive as well. In 7.2 **P** aborts **C1**, but before that process can complete it must find all of **C1**'s children and abort them first; thus **C2** actually gets freed before **C1**. For deep enough aborts it may be worth offloading an abort task to a separate processor to minimize the impact on **P**. Finally, in-order speculation introduces complexity with respect to buffering pre-existing frames. **C2** pops **d'** in 7.3 and gets **b"** not from **b** in **P** but from **b'** in **C1**. This requires a backwards pointer from **C2** to **C1** in the VM for efficient copying. Subsequently in 7.4 **C2** pops **b"** to return to **a'**, which must be copied from **a** in **C2**'s grandparent **P**. This operation is always deterministic, as lower frames on the stack execute later in program order, and in particular there is no way for **C1** to create a version of **a** nor for **P** itself to modify **a** while **C2** is still executing.

Figure 8 depicts the intricacies of commits under in-order nesting. In 8.1 **P** commits **C1**, inheriting all of its stack frames as usual, but also inheriting **C2** which is attached to **c**. Instead of freeing **C1**, **P** puts it on a 'garbage' list for later collection when all of **C1**'s children are dead. We need only one such backwards-pointing list per parent. Each node on the list is kept until completion of a DFS over its child nodes. In 8.2 **P**

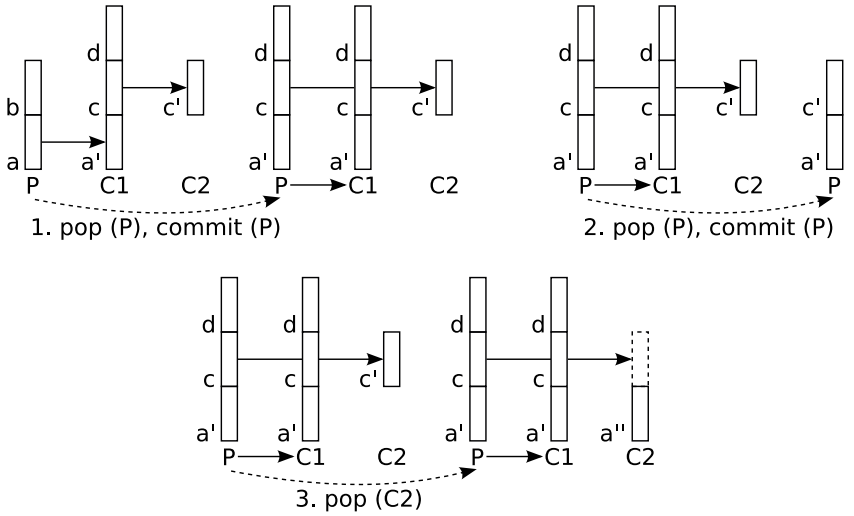

Figure 8: In-order nesting: commit, and pop from garbage.

now commits **C2** and gets **c'**. **C2** has no children and is immediately freed, and now that all of **C1**'s children are dead **C1** is also freed. The freelist scheme described in Section 2 was motivated by the need to handle **C2**'s memory: **C1** allocates **C2** on processor 2, some helper/worker thread executes **C2** on processor 3, and finally **P** frees **C2** on processor 1. The reason **C1** must be kept on a garbage list until **C2** is dead is illustrated in 8.3. **C2** pops **c'** and gets **a"** from **a'**. However, the source **a'** is always located in **C1**, even when **C1** has been committed by **P** as in the figure. This eliminates a race condition where **C2** pops **c'** and gets a pointer to **C1**, **P** commits and frees **C1**, and then **C2** attempts to copy the memory from **C1**. The alternative is to synchronize on lazy frame copying which is too expensive.

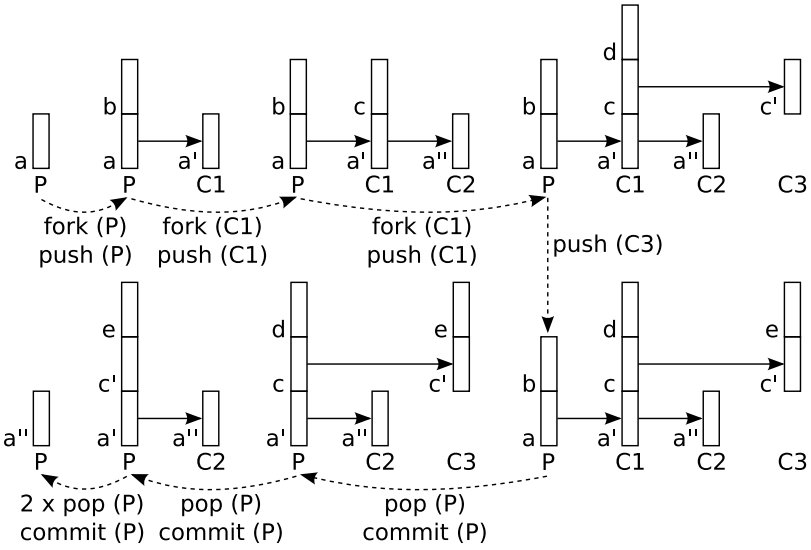

Figure 9: In-order and out-of-order nesting combined.

Figure 9 uses multiple patterns from Figures 4–8 to illustrate a combination of in-order and out-of-order nesting. The steps taken are 4.4, 7.1, 6.1, 5.1, 8.1, 6.4, 4.2, and 8.2. Not shown is that after 8.1 **C1** is kept on a garbage list until **C3** and **C2** are freed. We classify this speculation as having *nesting depth* 2 and *nesting height* 2 for its two levels each of in-order and out-of-order nesting respectively.

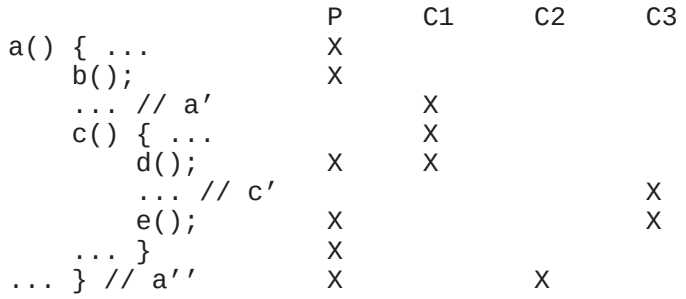

Figure 10: Sequential execution order of Figure 9.

Figure 10 depicts the normal sequential execution order of the code executed in Figure 9 and its mapping to non-speculative parent thread and speculative child tasks. An X in any of **P**, **C1**, **C2**, or **C3** indicates that the corresponding portion of the execution was executed by that thread or task. Note that there may be an X in **P** at the same time as any child, corresponding to a join, commit, and transfer of control in that stack frame.

4 Related Work

Speculative multithreading is relatively well-studied from a hardware perspective; Kejariwal and Nicolau maintain an extensive bibliography [13]. A general problem in SpMT is deciding where to fork speculative child tasks. Many systems operate at the loop level, some operate at the basic block level, and a few operate at the method level. We focus on creating child tasks at the method level for Java [21–25]. This appears appropriate since short methods and frequent invocations are idiomatic in Java programs [5, 11, 28, 30]. However, we also believe that well-structured programs written in other languages could benefit from method level speculation, and that our designs are not only applicable to Java. Method level speculation can also subsume loop level speculation [5], as described for Figure 7.1, albeit with extra invocation overhead.

There is prior work on both in-order and out-of-order nested speculation. Renau *et al.* extend a model with unlimited nesting *depth* to allow unlimited nesting *height* [26]. This contrasts with our work that began with unlimited nesting height [24] and now supports unlimited nesting depth. They propose a hardware architecture based on timestamps that is fairly complex and does not translate easily to software. Our model is designed for software SpMT and exploits a nearly universal program structure—the call stack—to ensure correctly ordered commits.

Our stack abstraction also provides a simple framework for understanding and unifying method level speculation approaches. For example, Whaley and Kozyrakis evaluate heuristics for Java method speculation, claiming to allow speculative threads to create speculative threads, i.e. in-order nesting [30]. However, all of their examples actually demonstrate out-of-order nesting. Goldstein *et al.* provide an efficient implementation of *parallel call* that uses a stack abstraction dual to the one presented here: the child task executes the method and the parent thread executes the continuation [9]. Pillar is a new language that supports this abstraction [1]. Although parallel call was not designed to be speculative, the speculation rules of our system could nevertheless be translated straightforwardly. Zhai briefly describes stack management for speculation [33], but does not provide details on the complexities of entering and exiting stack frames speculatively. Zahran and Franklin examine return address prediction in a speculative multithreading environment [32], and later consider entire trees of child tasks [31].

Osborne developed speculative computation for Multilisp [20]. The purpose of speculative execution in that

context is somewhat different: instead of aborting speculative computations because they are incorrect, the computations are aborted because they are unnecessary, and the abortion is a way to reclaim computation resources.

Mattson, Jr. found that speculative evaluation in Haskell can be supported with low overhead [17]. Ennals and Peyton Jones present a similar optimistic execution system that works together with the lazy evaluation model in Haskell [8]. Harris and Singh later used runtime feedback to drive speculation in Haskell with good results [10]. All of these speculation models might benefit from being described using our stack abstraction, and in turn our stack abstraction would become more robust if we extended it to accomodate them.

Our mechanism for recycling child task data structures is directly inspired by the Hoard model [2], which uses per-processor and global heaps to bound memory consumption and avoid false sharing. Michael later provide a lock-free allocator based on Hoard that offers a significant improvement [18], and Schneider *et al.* demonstrate another recent scalable malloc implementation [27]. Dice and Garthwaite also provide a mostly lock-free malloc that is 10 times faster than Hoard as originally published in some cases [6].

Somewhat heretically [3], we have created a specialized lightweight Hoard-like custom memory allocator that differs in two ways from the original Hoard publication: it uses per-thread blocks, called superblocks, rather than per-processor blocks, and it recycles entire ownership dominator trees, maintaining links across a library interface. The key assumption we make is that there is one thread per processor. This is suitable for our constrained speculation machinery which takes single threads and parallelizes them, and will not speculate at all if all processors are occupied by non-speculative threads. Of course, the Hoard software has since evolved to accomodate initial performance concerns, including in particular by supporting per-thread sub-heaps. Thus the recycling of entire structures at once is expected to be the only real advantage of our system in a side-by-side comparison.

Boyapati *et al.* combine user-specified ownership types with region-based memory management [4]. Lattner and Adve later provide a system for automatic pool or region allocation that segregates the memory required by an individual data structure into its own pool [15]. Mitchell subsequently examines the runtime structure of object ownership for many large real-world applications, identifying many dominator trees [19]. Our experience suggests it would be interesting to combine the different ideas from these works to create a general purpose multiprocessor allocation system that returns usable pre-assembled data structures.

5 Conclusions & Future Work

The performance and correctness issues in speculative multithreading are all memory-based. Our first contribution is a simple multithreaded custom allocator for child task data structures that relies on knowledge about ownership dominator trees. Although it eliminates major performance bottlenecks in our system, we have not actually experimented with state-of-the-art multithreaded allocators in a controlled comparison. Our intuition is that although the synchronization and memory locality in our scheme are probably sub-optimal, the 37-fold reduction in calls to malloc and free more than compensates. We would be quite excited to see recycling of aggregate data structures evolve into a general purpose memory management paradigm.

Our second contribution is a clean, simple, and comprehensive stack based abstraction for method level speculation. As future work, one alternative is to fork and join children at method entry and exit points instead of at callsites [30]. Our model can accomodate this with minor modifications. Another issue is that our model assumes references to stack variables cannot be passed to callee frames. Although definitely a problem for C programs, JVMs are also permitted to allocate a Java object on the stack and this can cause

synchronization errors. If some child copies a parent frame with a stack allocated object, the parent thread modifies that object, and then the parent commits the child, this will erase the modification. As for potential for compiler assistance, our implementation copies entire stack frames between processors, whereas only those local variables live after invocation need copying. Furthermore, a compiler could pack the copied locals together for efficiency.

Finally, we would like to provide a small-step operational semantics for our stack model and use it to prove that a concurrent speculative multithreading algorithm operating on our abstraction satisfies various correctness properties. We would also like to unify our abstraction with the parallel sequential call one used by Pillar in order to transfer results. As more general challenges, speculation past monitor operations introduces significant complexity [29], as does speculative object allocation [12] and broader reconciliation with transactional execution [14].

We have built a prototype system consisting of SableSpMT [22–24] and libspmt [25] that supports everything we describe for Java programs. However, the performance indications in this paper are qualitative and await deeper characterization. We recently summarized our work on SpMT to date and major directions for future work [21].

Acknowledgements

This research was funded by the IBM Toronto Centre for Advanced Studies and the Natural Sciences and Engineering Research Council of Canada. This report was originally submitted to MSPC'08 in November 2007. We would like to thank the referees for their helpful comments, questions, and suggestions.

References

[1] T. Anderson, N. Glew, P. Guo, B. T. Lewis, W. Liu, Z. Liu, L. Petersen, M. Rajagopalan, J. M. Stichnoth, G. Wu, and D. Zhang. Pillar: A parallel implementation language. In *LCPC'07: Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing*, volume 5234 of *LNCS: Lecture Notes in Computer Science*, pages 141–155, Oct. 2007.

[2] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, Nov. 2000.

[3] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *OOPSLA'02: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–12, Nov. 2002.

[4] C. Boyapati, A. Salcianu, J. William Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *PLDI'03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 324–337, June 2003.

[5] 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.

[6] D. Dice and A. Garthwaite. Mostly lock-free malloc. In *ISMM'02: Proceedings of the 3rd International Symposium on Memory Management*, pages 163–174, June 2002.

[7] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI'07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 223–234, June 2007.

[8] R. Ennals and S. P. Jones. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In *ICFP'03: Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, pages 287–298, Aug. 2003.

[9] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy threads: Implementing a fast parallel call. *JPDC: Journal of Parallel and Distributed Computing*, 37(1):5–20, Aug. 1996.

[10] T. Harris and S. Singh. Feedback directed implicit parallelism. In *ICFP'07: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming*, pages 251–264, Oct. 2007.

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

[12] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. McRT-Malloc: a scalable transactional memory allocator. In *ISMM'06: Proceedings of the 2006 International Symposium on Memory Management*, pages 74–83, June 2006.

[13] A. Kejariwal and A. Nicolau. Speculative execution reading list. `http://www.ics.uci.edu/~akejariw/SpeculativeExecutionReadingList.pdf`, 2007.

[14] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, Dec. 2006.

[15] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI'05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 129–142, June 2005.

[16] D. Lea. A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc`, Apr. 2000. First published in 1996.

[17] J. S. Mattson, Jr. *An effective speculative evaluation technique for parallel supercombinator graph reduction*. PhD thesis, University of California at San Diego, La Jolla, California, USA, 1993.

[18] M. M. Michael. Scalable lock-free dynamic memory allocation. In *PLDI'04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 35–46, June 2004.

[19] N. Mitchell. The runtime structure of object ownership. In *ECOOP'06: Proceedings of the 20th European Conference on Object-Oriented Programming*, volume 4067 of *LNCS: Lecture Notes in Computer Science*, pages 74–98, July 2006.

[20] R. B. Osborne. Speculative computation in Multilisp. In *LFP'90: Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 198–208, June 1990.

[21] C. J. F. Pickett. Software speculative multithreading for Java. In *OOPSLA'07 Companion: Companion to the Proceedings of the 22nd ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications*, pages 929–930, Oct. 2007.

[22] C. J. F. Pickett and C. Verbrugge. Return value prediction in a Java virtual machine. In *VPW2: Proceedings of the 2nd Value-Prediction and Value-Based Optimization Workshop*, pages 40–47, Oct. 2004.

[23] 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.

[24] C. J. F. Pickett and C. Verbrugge. Software thread level speculation for the Java language and virtual machine environment. In *LCPC'05: Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, volume 4339 of *LNCS: Lecture Notes in Computer Science*, pages 304–318, Oct. 2005.

[25] C. J. F. Pickett, C. Verbrugge, and A. Kielstra. libspmt: A library for speculative multithreading. Technical Report SABLE-TR-2007-1, Sable Research Group, School of Computer Science, McGill University, Mar. 2007.

[26] 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.

[27] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *ISMM'06: Proceedings of the 2006 International Symposium on Memory Management*, pages 84–94, June 2006.

[28] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *OOPSLA'05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 439–453, Oct. 2005.

[29] A. Welc, S. Jagannathan, and A. L. Hosking. Revocation techniques for Java concurrency. *CC:PE: Concurrency and Computation: Practice and Experience*, 18(12):1613–1656, Oct. 2006.

[30] 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.

[31] M. Zahran and M. Franklin. Dynamic thread resizing for speculative multithreaded processors. In *ICCD'03: Proceedings of the 21st International Conference on Computer Design*, pages 313–318, Oct. 2003.

[32] M. M. Zahran and M. Franklin. Return-address prediction in speculative multithreaded environments. In *HiPC'02: Proceedings of the 9th International Conference on High Performance Computing*, pages 609–619, Dec. 2002.

[33] A. Zhai. *Compiler optimization of value communication for thread-level speculation*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, Jan. 2005.