McGill University
School of Computer Science
Sable Research Group

libspmt: A Library for Speculative Multithreading

Sable Technical Report No. 2007-1

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

March 12th, 2007

Abstract

Speculative multithreading (SpMT), or thread level speculation (TLS), is a dynamic parallelisation technique that uses out-of-order execution and memory buffering to achieve speedup. The complexity of implementing software SpMT results in long development lead times, and the lack of reuse between different software SpMT systems makes comparisons difficult. In order to address these problems we have created libspmt, a new language independent library for speculative multithreading. It provides a virtualization of speculative execution components that are unavailable in commercial multiprocessors, and enables method level speculation when fully integrated. We have isolated a clean and minimal library interface, and the corresponding implementation is highly modular. It can accommodate hosts that implement garbage collection, exceptions, and non-speculative multithreading. We created libspmt by refactoring a previous implementation of software SpMT for Java, and by aiming for modularity have exposed several new opportunities for optimization.

1 Introduction

libspmt is a new library designed to simplify the task of implementing speculative multithreading in software. *Speculative multithreading* (SpMT), or thread level speculation (TLS), is an aggressive and optimistic parallelization scheme that operates dynamically on non-parallel code. Many proposals focus on hardware architecture implementations [3, 7, 33], but numerous software approaches have also been attempted [4, 8, 20, 25, 28, 29].

The salient features of speculative multithreading systems include: 1) support for memory access buffering or logging; 2) some mechanism to detect violations and either undo or prevent unsafe operations; and 3) a means to either commit the speculative execution in a manner that preserves original program semantics, or abort the execution safely. A basic parallelization strategy is required: speculation, or out-of-order execution, may occur at any or all of the basic block [3], loop [4, 8, 20, 25, 33], method [7, 28, 29, 37], or lock [36] levels. Perhaps most importantly, the parallelization occurs at the thread level as opposed to the instruction level, and requires two or more CPUs, cores, or virtual cores for speedup. Although there exists variance between the parallelization strategies, there is also considerable commonality.

SpMT in fact shares many features with other designs, most notably transactional programming [19], but also rollback for debugging [11] and checkpointing [9, 34]. Transactional execution of atomic regions [5, 17, 18, 30, 31] is roughly equivalent to lock-based speculative multithreading [36], in which a thread may enter and execute a critical section speculatively. The user model for transactional programming and lock-based speculation differs from that for speculative multithreading, however. In transactional programming and lock-level speculation, the user creates threads and relies on the compiler, runtime system, or hardware to handle optimistic execution of critical sections or atomic regions. This contrasts with speculation over sequential code, which does not require any source level changes, but must address the additional challenge of creating and managing new speculative threads.

We introduce libspmt as a modular and core component of an SpMT system. It extracts, simplifies, and virtualizes the main features required for speculative multithreading and related designs. In particular, it supports method level speculation, in which parents fork child continuations at method invocations and join them upon returning from the call. The specific motivation for this work is the need to enable speculation in IBM's production Java JIT compiler; previously we invested considerable effort in implementing software SpMT correctly for Java [29], and we set out to modularize and reuse that code base and design. Maintaining a fully functional client alongside the development of libspmt helped drive many of the design decisions.

The general benefit is that we have now defined a relatively minimal and clean interface to a specialized virtual machine or runtime system that can be used to enable speculation in a wide variety of hosts; we also have a robust implementation of that interface, and a client that uses it, namely a refactored version of our initial Java bytecode interpreter system, SableSpMT [28]. These contributions will reduce the complexity of implementing and optimizing software speculative multithreading, facilitate comparisons between approaches, and in our particular case allow us to pursue JIT-based optimizations.

libspmt acts as a specialized architecture simulation or virtual machine providing speculative execution facilities that other compilers and virtual machines can target. It provides a virtualization of previously proposed hardware extensions, much as software transactional memory libraries provide a virtualization of hardware transactional memory architecture extensions. It differs from traditional simulations in that there is no straightforward mapping of the design to a hardware circuit. However, it enables real hardware to perform speculative execution, and as an experimental research tool runs significantly faster than an actual simulator [21]; our present and naïvely optimized interpreter-based client exhibits 5x slowdowns but up to 2x relative speedups [28]. Additionally, as hardware support starts to become available [16], libspmt can serve as portable layer providing access to it.

libspmt is also designed as a separate system component, much like a reusable garbage collector, malloc implementation, or general purpose multithreading library, albeit with significantly more complex usage requirements. We have isolated only the language-independent logic in an effort to increase system modularity, and to make that logic available through an API to compilers, interpreters, and virtual machines for different languages. Of course, a completely independent library is also ostensibly more testable and maintainable.

libspmt can be used and extended in two different contexts. First, it can be used as a complete runtime system for enabling speculation, specifically method level speculation. In this respect it can be extended in the future to support other modes of speculation when fully integrated into other systems. Second, it can be used as a repository of modules for individual reuse and experimentation. In this respect it can be extended by adding new modules and exchanging implementations, for example by adding new value predictors or replacing our dependence buffer with a highly-optimized software transactional memory library.

1.1 Contributions

We make the following specific contributions:

- The design of a modular library for speculative multithreading. libspmt provides advanced facilities for software method level speculation in relatively arbitrary execution environments.
- A clean and minimal interface design that provides a separation of tasks without degrading performance.
- A modular library implementation that allows components to be reused or replaced in different contexts. We illustrate this modularity through an easy implementation of a *virtual* single-threaded debugging mode. This modularity also exposes new opportunities for optimization.
- Support for various complex host or client features, including garbage collection, exception handling, and non-speculative multithreading.
- A newly refactored version of SableSpMT, a Java client that uses libspmt. It now serves as both an API reference and functional testing program. Both SableSpMT and libspmt are available under the LGPL.

In the next section we present the interface to libspmt, providing details on its usage. In Section 3 we describe the library implementation, and in Section 4 our development process. Section 5 presents related work on speculative multithreading, and we conclude and describe future work in Section 6.

2 Interface

We first present the interface to libspmt, as depicted and described in Figure 1. We have tried to minimize the libspmt interface, so as to increase flexibility in modifying both the *host* code that uses it and the library code that implements it.

In the remainder of this section we define some necessary terminology, discuss prerequisites for using the library, and then describe broad use of the library, the specifics of forking and joining speculative tasks, the details of support for speculative execution, and finally how the API supports some optional features: non-speculative multithreading, garbage collection, and exception handling.

2.1 Overview

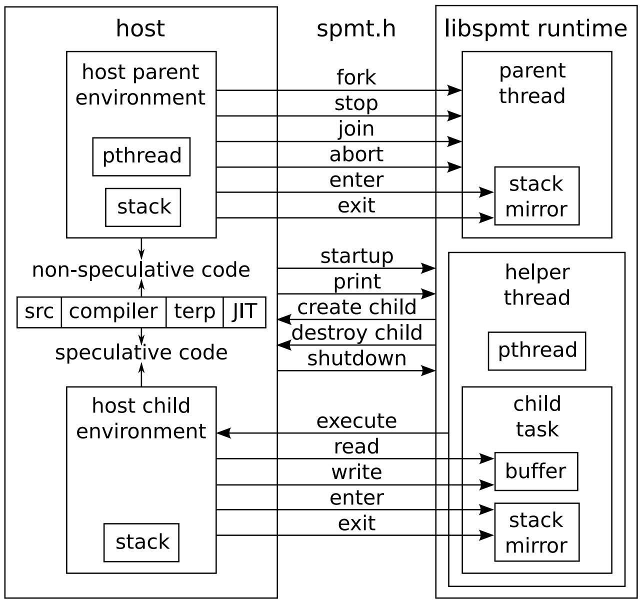

Figure 1: *Interface to libspmt.* A host is any client of libspmt, and may consist of source code, compiled code, an interpreter, or a JIT compiler. It communicates with the libspmt runtime using a set of calls and callbacks defined in `spmt.h`. The host is responsible for startup and shutdown of the runtime, and can optionally print statistics aggregated over execution. At startup, libspmt creates helper threads for execution of child tasks, and at shutdown will destroy these threads. The host modifies its native threads to become parent threads that fork, stop, join, and abort child tasks. The host must provide callbacks for creating and destroying host-specific child task environments, speculative versions of non-speculative code, and a callback that allows children to execute this code. A child executing in the host protects non-speculative main memory by using a local execution stack for automatics and a dependence buffer in libspmt for heap and global data. Both children and parents enter and exit methods in the host, and the operations are mirrored in libspmt. The host is responsible for buffering and committing local child stacks, whereas other thread control operations are managed by the libspmt runtime.

In order to describe libspmt efficiently we need to define some terminology, which also corresponds to module names. A *host* is the user or client of libspmt, so called because it literally “hosts” the virtualized SpMT logic, and because the relationship involves callbacks. The host may interact with the library as a result of compiler transformations, virtual machine extensions, or source code modification. The libspmt *runtime* is a global singleton for an entire libspmt instance, through which all other libspmt data structures are reachable. This is not to be confused with the host, which in the case of a virtual machine may also be considered a runtime unto itself. A *thread* is bound to an actual OS-level thread that is scheduled by the kernel. libspmt uses pthreads, and there is currently a requirement for the host to use pthreads as well. However, if the host can provide the right callbacks, libspmt could fairly easily be modified to use a generic and portable wrapper around OS threads. A *parent* is a host thread that executes non-speculative code. There is a one-to-one mapping between host threads and parents. A *child* is a task that will be executed speculatively. Under speculative method level parallelism, it is a continuation, or the sequence of instructions following a method call. Parent threads *fork* children by enqueuing child tasks on a global libspmt priority queue. When a parent thread returns from a method call, it attempts to *join* a child task if it forked one. A *helper* is a libspmt thread that dequeues child tasks from the priority queue and executes them. There is a one-to-one mapping between libspmt threads and helpers. Helper threads are *not* simply prefetching assist threads, although they could actually be used for that purpose by executing child tasks and ensuring that they never commit.

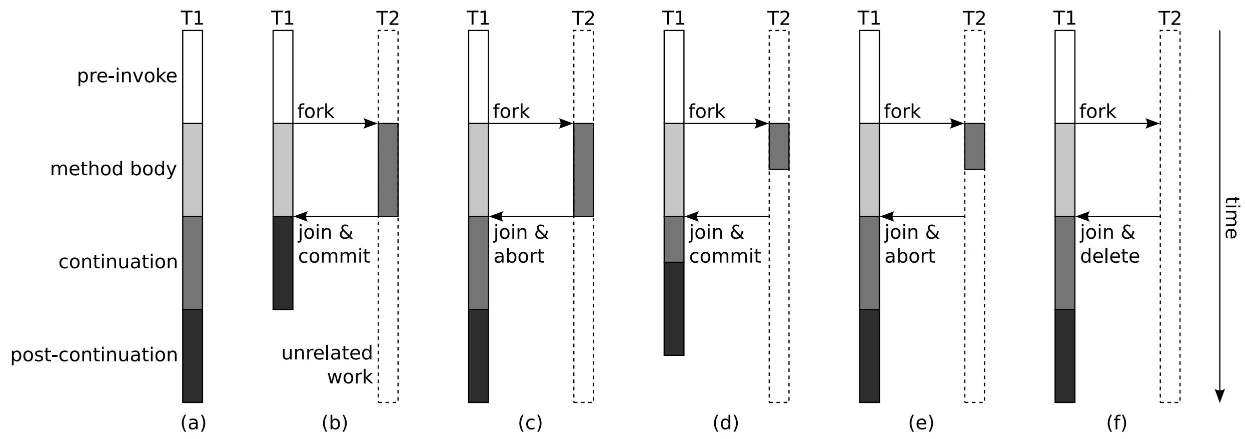

Figure 2: *Thread control.* libspmt supports method level speculation, in which children execute method continuations. **(a)** T1 is non-speculative and executes a pre-invoke sequence of instructions, a method body, a method continuation, and a post-continuation sequence of instructions. **(b)** T1 forks a child task before executing the method body, enqueuing it on a priority queue. T2 is a helper or worker thread that dequeues and binds itself to the child task in order to execute the method continuation speculatively. All reads from and writes to main memory are buffered, and unsafe operations are prevented. T1 eventually returns to its invocation point and joins T2, signalling it to stop execution of the child. T2 immediately proceeds to execution of other unrelated work. T1 validates the child by comparing all of its reads against main memory, and then commits the result by flushing all of its writes and copying over child stack frames. T1 jumps to execution of the post-continuation instructions. **(c)** Same as (b), except that validation fails and T1 must abort the child, and then execute the continuation instructions again. **(d)** Same as (b), except that T2 stops execution of the child without receiving a signal from T1. This may be due either to execution of an illegal instruction, or to reaching some predefined limit on child length. After the commit, T1 must complete execution of the continuation sequence that the child could not. **(e)** Same as (d), except that validation fails, and T1 must execute the entire continuation. **(f)** T1 returns to the method invocation point before T2 is able to dequeue the child task, and so T1 joins the child simply by deleting the task from the priority queue. T2 is not involved in the process.

2.2 Prerequisites

The host must provide several resources in order to interface effectively with libspmt. These include an optional compilation component, speculative versions of non-speculative code, and a set of callbacks for handling child tasks.

Compilation. SpMT implementations typically involve a compiler for either transformation or analysis. For purposes of transformation, libspmt is a high-level virtual target for which a compiler can generate calls. For analysis, specific results can be communicated as arguments to runtime calls. There is no compiler in libspmt, and indeed it has no concept of either intermediate or executable code as it is language independent; all of the actual speculation takes place in the host environment.

As an example, Soot [35] is the compiler framework we use for SableSpMT. We have an ahead-of-time analysis that detects statically whether or not a return value is actually consumed [26]. This is communicated to SableSpMT via classfile attributes. These attributes are parsed, and liveness information for individual return values is sent to libspmt. libspmt uses this information to determine whether or not return value predictions should be made.

Speculative Code Generation. The host must provide speculative versions of non-speculative code which libspmt can use to execute child tasks. The three scenarios for generating code correspond to the three primary usage scenarios: 1) in the case where the host is a virtual machine interpreter, this involves duplicating the code array, patching unsafe instructions with speculative versions, and fixing up jumps; 2) in the case where the host is a compiler and the code it generates, the host creates an alternate version of the binary code to be executed, and ensures that speculative code can safely switch execution back to the corresponding non-speculative code; 3) in the case where the host is actually source code with calls to the library API, the user manually ensures that the speculative code to be executed performs the same function as the original non-speculative code. In all three cases the speculative code and non-speculative code can actually overlap if protected with checks for speculative execution. However, it is generally more efficient to create a duplicate and specialized mirror version of the non-speculative code for speculation.

Callbacks. Finally, the host must provide three callbacks that libspmt will use to control child operations:

- `spmt_host_child_(create|destroy)()`: These functions create and destroy a host-specific thread environment that will be used for speculative execution. This must include memory for a call stack. Once allocated and initialized, the memory is managed by libspmt, which calls `spmt_host_child_-create()` in one thread and `spmt_host_child_destroy()` in another. The thread environment returned must be hidden from the host. This means that it cannot be visible to language level reflection capabilities, and can neither trigger nor be traced by a garbage collector; it generally cannot be accessed in any way if non-speculative semantics are to be preserved.
- `spmt_host_child_execute()`: This function executes a child task. A helper thread will call this function after dequeueing and binding itself to some child task. The call takes a host thread environment and predicted return value as arguments. The implementation must first restore the non-speculative state that was previously saved in the host child thread environment, as discussed in Section 2.4. It then proceeds to make the state speculative and locate the host-provided speculative code for execution. Finally it sets the return value if the method continuation follows a non-void method call, and jumps directly to speculative execution of the child.

These operations are implemented as callbacks because they provide libspmt with decoupled access to the host environment and help to minimize the interface.

2.3 Runtime Usage

There are various ways to control the behaviour of the libspmt global runtime. A call to `spmt_runtime_create()` instantiates and returns a pointer to a singleton runtime object. Function pointers are then used to register the mandatory callbacks. The host can also set various final runtime options. These include a single-threaded mode, verbose logging, the number of helper threads [28], and the ability to disable support for individual aspects of speculation [29]. Once initialized, a call to `spmt_runtime_startup()` will create the actual helper threads that wait for children to be enqueued.

A call to `spmt_runtime_shutdown()` joins all helper threads. If statistics gathering is enabled, `spmt_runtime_print()` will print a summary of execution to standard error. Finally, `spmt_runtime_destroy()` will free all library memory. It is generally recommended that each of these runtime functions be called once per program execution, as creating and joining OS level pthreads and allocating and freeing necessary runtime memory are expensive operations; additionally, this prevents unnecessary destruction and recreation of runtime profile information.

2.4 Thread Control

libspmt supports a wide range of options for hosts to control the speculative execution model, as shown in Figure 2. Speculation *probes* are used as the main speculation points at which child forking, stopping, and joining occur.

Speculation Probes. A host uses *probes* to communicate host-determined locations for speculation to libspmt. These are tied to method invocations, and the host is required to identify them properly, as they are completely implementation and language dependent. There are three kinds of probe, and they are best described by considering a program callgraph.

The *callgraph* of a program is a directed bipartite graph between methods and callsites. An edge from a method to a callsite indicates that the callsite is *contained* within the body of the method. A method may contain any number of callsites, but a callsite may only be contained within a single method. An edge from a callsite to a method indicates that the callsite *invokes* the method. A callsite may invoke any number of methods, and a method may be invoked at any number of callsites. A *callsite probe* captures a callsite and all of the methods it invokes. A *method probe* captures a method and all of the callsites that invoke it. A *methodcall probe* captures a callsite and one method that it invokes; equivalently, it captures a method and one callsite that invokes it. These three probe types are illustrated in Figure 3.

A host registers a probe by passing its host address to libspmt, and in return receives the handle of a newly created libspmt probe object. The host then uses this handle to fork and join threads, and libspmt uses the object to collect data.

There are different tradeoffs for each kind of probe. In our experiments, we found that there are always more callsites in a program than methods, both statically and dynamically. This leads to increased memory consumption and less sharing of data when callsite probes are used. There are more edges from callsites to methods than either callsites or methods, but since most callsites are dynamically monomorphic [10], there is usually not a large difference between a callsite probe and a methodcall probe. An advantage of

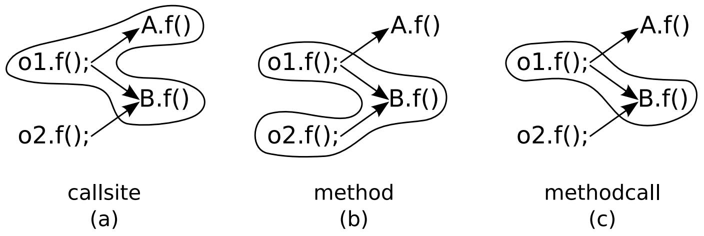

Figure 3: *Speculation probe types.* `o1.f()` is a polymorphic callsite that may invoke either of the `A.f()` or `B.f()` methods. `o2.f()` is a monomorphic callsite that invokes only `B.f()`. Shown are the three different probe types that each capture the invoke from `o1.f()` to `B.f()`.

callsites in an interpreter, where method invocations and returns are comparatively expensive, is that the child can begin execution before the invoke and continue until after the return. In the case of return value prediction, a memoization predictor is most naturally associated with a method probe, as it can then benefit from population by all invoking callsites. However, a context predictor associated with a method probe suffers from pollution if more than one callsite contributes to its value history.

Child Forking. When a parent thread in the host reaches a callsite, method, or methodcall for which there is an associated probe, it attempts to fork a child:

```
child = spmt_parent_request_fork (parent, probe);
if (child != NULL)
  {
    host_child = spmt_child_get_host_child (child);
    /* save just enough thread state */
    ...
    spmt_parent_complete_fork (parent, child);
  }
```

It first requests that a fork be carried out, and libspmt returns a non-NULL child object if the internal probe profile suggests that forking will be beneficial. The host then retrieves a host-specific child environment, allocated previously by an `spmt_host_child_create()` callback, and uses it to save the current thread state. A final call to libspmt tells it to complete the fork by enqueueing the child task.

Child Stopping. A child *stops* when it is finished speculative execution; this occurs before joining and the subsequent abortion or committal. There are three events which trigger a child stop: 1) attempting to execute an unsafe instruction; 2) reaching a predefined limit on child length; 3) a request from a parent thread. A child stops by saving the current state and returning to `spmt_host_child_execute()`, which in turn returns control to libspmt.

Speculative code must force a child to stop before executing unsafe instructions. It must also ensure that the child periodically and regularly calls `spmt_child_stop_requested()`; if the host periodically polls for garbage collection this may be co-opted as an appropriate place for the check. This function will return true after the child has reached a certain length, measured either by counting number of calls to this function or by using a processor timestamp. It will also return true if the parent thread has called `spmt_parent_-request_child_stop()`; this happens when the parent anticipates reaching a join point in the near future, giving the child time to stop while the parent completes execution of its method.

Child Joining. Joining is the process of waiting for a child to stop, validating its results, and committing or aborting the speculative execution. When a parent thread in the host returns from a method or methodcall, or returns to a callsite, and there is an associated probe, it attempts to join a child:

```
child = spmt_parent_request_join (parent, ret_val);
if (child != NULL)
  {
    host_child = spmt_child_get_host_child (child);
    /* load final child state */
    ...
    spmt_parent_complete_join (parent, child);
  }
```

It first requests that a join be carried out, and libspmt returns a non-NULL child object if there is a child attached to the calling stack frame, as discussed in Section 2.5, and if the validation and commit process internal to libspmt is successful. This call involves a busy wait if the child has not stopped. The host then retrieves the host-specific child environment, and uses it to restore the final child state and switch to non-speculative execution.

2.5 Speculative Execution

Execution of speculative code must be safe, and hidden from non-speculative threads in the host; most importantly, it cannot affect the non-speculative execution semantics, which include the host threading model. If speculative code encounters an unsafe operation, it must stop execution immediately and await joining, as discussed in Section 2.4. There may be numerous safety and correctness issues involved [29]. In the remainder of this section we describe two essential components of speculative execution and how libspmt supports them, namely dependence buffering and stack buffering.

Dependence Buffering. The libspmt dependence buffer module allows a speculative child to safely execute past reads from and writes to main memory; a first approximation considers these operations unsafe and they force speculation to stop immediately. Each child has a buffer with two public operations, `spmt_buffer_read (buffer, address, size)` and `spmt_buffer_write (buffer, address, value, size)`. The size argument allows the host to buffer 1, 8, 16, 32, or 64-bit values. All other buffer operations, such as validation and committal, occur within libspmt at child join time.

As a particular implementation detail that our interface exposes, there are two problems with using the dependence buffer if union types with heterogenous field sizes may be accessed by speculative code. Strongly-typed languages such as Java and C# do not permit such unions, but weakly-typed ones such as C and C++ do.

First, if a write occurs to a memory location already contained in the buffer then that value is simply updated. This means that writes are not necessarily committed in the exact order they were written. A call to `problem_one()` followed by a commit will leave `heap.a` with the value `0x00220033` instead of `0x00000033`, as the first write will reserve a place in the commit queue for `heap.a`, and `heap.a` will be committed before `heap.b`. This could be fixed by numbering all writes and sorting the buffer before commit.

Second, different buffer slots are used for `heap.a` and `heap.b[0]` through `heap.b[3]`, even though `&heap.b[0]` equals `&heap.a`, because value size is also used to detect collisions. A call to `problem_two()` will thus return `0x00000011` instead of `0x00000022`. Furthermore, since the read is not separately buffered, as the value is already contained in the write buffer, the buffer will successfully validate and commit, hiding the read-after-write dependence violation.

```
union {
  spmt_u32_t a;
  spmt_u8_t b[4];
} heap;

void problem_one (spmt_buffer_t *x) {
  spmt_buffer_write (x, &heap.a, 0x11, SPMT_U32);
  spmt_buffer_write (x, &heap.b[2], 0x22, SPMT_U8);
  spmt_buffer_write (x, &heap.a, 0x33, SPMT_U32);
}

spmt_u32_t problem_two (spmt_buffer_t *x) {
  spmt_buffer_write (x, &heap.a, 0x11, SPMT_U32);
  spmt_buffer_write (x, &heap.b[0], 0x22, SPMT_U8);
  return spmt_buffer_read (x, &heap.a, SPMT_U32);
}
```

The current solution to these problems is to restrict hosts that allow for union types and overlapping memory accesses to using only word-aligned buffer calls. The process is analogous to that used by a hardware cache when writing subwords. In order to write to `heap.b[2]`, the host must: 1) compute the address of `heap.-b[2]`'s word; 2) create a bitmask using a call to `spmt_buffer_read()`, the offset of `heap.b[2]` in the word, and the size of `heap.b[2]`; 3) or together the bitmask with an appropriately shifted value for storage; 4) and then finally call `spmt_buffer_write()` with this computed value and the word-aligned address.

Stack Management. A general problem in speculative multithreading is that shared call stacks may require an expensive buffer call for each access of a local or automatic variable. The solution libspmt provides is wholesale buffering of stack frames upon method entry and exit, and it allows for unrestricted speculative modification of a local stack; accordingly, each child must be provided with a separate execution stack. Upon forking, the current parent frame is copied to the child. When a new method is invoked, a new stack frame is simply pushed. When returning to a method that the child did not invoke, the frame must be copied from the parent; this is safe, because the parent is guaranteed to join a child before modifying any stack frames pushed prior to the join point. The host is responsible for all stack buffering and copying. This provides room for compiler optimization, and obviates the need for library calls to handle what amounts to a `memcpy()`.

The other significant complication of stack management is that of storing per-invoke information. libspmt associates certain information with stack frames: a parent stack frame contains a child object if one was forked in that frame, and a timer object for measuring invocation times. It also contains pointers to the relevant source callsite and target method speculation probes. In order to minimize changes necessary to host data structures, libspmt maintains a mirror of the stack to manage this information; it does so by requiring notification from both parents and children when stack frames are entered and exited. These mirror stacks also allow the libspmt runtime to traverse stacks independently of the host.

2.6 Optional Host Features

The libspmt interface provides mechanisms to support complex host components found in modern execution environments such as Java virtual machines. These include non-speculative multithreading, garbage collection, and exception handling.

Multithreading. Unchecked, multithreaded hosts will compete with libspmt for CPU resources. Accordingly, libspmt provides a flexible mechanism that gives priority to host threads: it is always better to execute a non-speculative thread than a speculative one.

For each operating system thread that the host creates as part of its normal execution, a corresponding library parent object must be created, even if speculation does not occur on that thread. `spmt_parent_create()` returns a pointer to a parent object that the host can store in the host parent thread environment. When the host thread is terminated, `spmt_parent_destroy()` is called.

Afterwards, any time a parent thread starts and stops execution, the host calls `spmt_parent_start()` and `spmt_parent_stop()` respectively. Reasons for starting include thread creation and wakeup, and reasons for stopping include thread destruction, sleeping, and blocking. This allows the runtime to ensure that the sum of running host parent and libspmt helper threads never exceeds the number of available processors, except if all threads are non-speculative.

Garbage Collection. Garbage collection is a feature of many modern languages, or at least language implementations, and is problematic in that it moves objects without updating dependence buffers; without special precautions, speculative object references are unsafe after GC. The host can quickly invalidate all of the children attached to a parent thread by calling `spmt_parent_abort_all_children()`; libspmt will then use the parent stack mirror to find attached children and abort them. A future experiment could involve updating child dependence buffers during GC, which may be worthwhile if collections are frequent enough.

Exception Handling. Another common feature of modern languages is exception handling, which is problematic in that it allows a method to return abruptly and to a location other than the callsite at which it was invoked. The consequence is that the parent thread might never return to its join point. The support mechanism for this in libspmt is similar to that for garbage collection: a given parent thread can abort any child attached to its stack frame at any time simply by calling `spmt_parent_abort_current_child()`; libspmt will then use the parent stack mirror to find the child attached to the current frame and abort it.

3 Implementation

The implementation of libspmt includes many modules whose behaviour and functionality is largely independent of the library interface. In this section we describe interesting aspects of the design, and where applicable, how they have evolved from their previous implementation. These include isolated statistics gathering, portable support for atomic operations and spinlocks, simple priority queueing, a new design for a *virtual* single-threaded mode, our refactored dependence buffering implementation, newly isolated value predictors, and a completely new memory management system for speculative child tasks.

An overview of the libspmt implementation is given in Figure 4. The runtime provides synchronized access to speculation probe tables, a memory manager, and a priority queue. Probe tables contain the probes that are

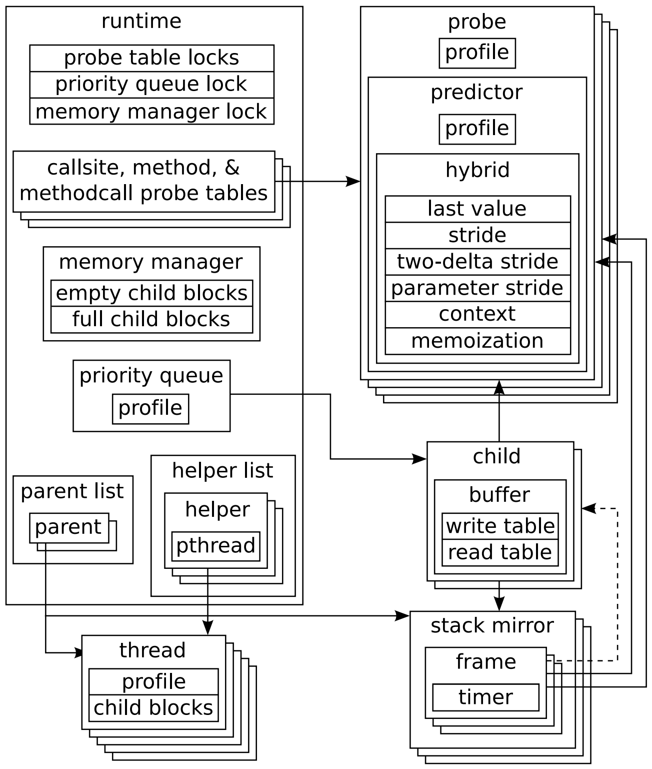

Figure 4: *Implementation of libspmt.* Shown are the dynamic relationships between the main data structures in libspmt.

used to fork and join child tasks. Each probe contains profiling information as well as a value predictor. The predictor maintains a profile and contains a hybrid predictor, which in turn contains various sub-predictors. Any sub-predictor may be used to replace the hybrid dynamically. The priority queue contains child tasks enqueued by parents that await execution by a helper. The memory manager contains a list of empty child blocks and a list of full child blocks. These are exchanged with thread-local full and empty child blocks respectively.

The runtime also maintains lists of all parent and helper threads created. Parent objects are bound to host parent threads, and helper objects are bound to libspmt helper threads. Each is associated with a common thread object that manages thread-local child blocks. Children are allocated and freed by thread objects, and each child points to the probe at which it was forked. A child contains a dependence buffer that is implemented by a write sub-buffer layered over a read sub-buffer. Both child and parent objects contain mirrors of the stacks associated with child and parent environments in the host. These mirrors represent stacks abstractly using a list of frames. Each frame has pointers to source callsite and target method probes, a timer for online profiling of method execution times, and in the case of parent objects, a pointer to any child forked in that frame. Profiles associated with the priority queue, speculation probes, predictors, and thread objects allow for both *post mortem* and online analysis. All memory is reachable from the runtime object and freed appropriately upon shutdown.

3.1 Statistics

libspmt is a complete speculative multithreading implementation capable of running advanced benchmarks. It can be used to collect a wealth of valuable profiling information for both online and *post mortem* analysis and optimization. Many of the analyses possible were previously tied up in our Java-specific SableSpMT implementation [28].

The priority queue object counts enqueue, dequeue, and delete operations for each priority. Threads gather profiling information using stopwatch timer objects, providing breakdowns of both useful work and overhead. Similarly, timers associated with stack frames measure child and parent execution lengths, necessary inputs to forking heuristics. Individual speculation probes aggregate data over all executions of the probe; these data are varied and can be used to calculate properties such as speculation success rates and prediction accuracies.

Additional statistics are gathered if libspmt is built with `--enable-stats`, which exposes the `spmt_runtime_print()` function that prints statistics to `stderr` after runtime shutdown. This function calls the corresonding `print()` function on globally reachable objects, including the priority queue, parent and helper threads, and all probes in the speculation probe registries.

3.2 Spinlocks and Atomic Operations

For performance reasons, libspmt uses spinlocks instead of pthread mutexes; this is possible because there is a one-to-one mapping between threads and processors. The only exception to this rule is when the number of parent threads exceeds the number of processors, as discussed in Section 2.6, in which case speculation is inactivated anyway. There are two kinds of spinlock, a simple test-and-test-and-set lock built directly on top of CAS or LL/SC, and more complicated CLH queue locks [22], where each thread spins on a separate cache line. In our current system, both achieve comparable performance. pthread mutexes and condition variables are indeed necessary to make helper threads sleep and wake when the number of running non-speculative threads changes, and also to control the single-threaded execution mode.

A few platform specific atomic operations must be provided in order for libspmt to function. Atomic compare-and-swap (CAS) or load-linked/store-conditional (LL/SC) are mandatory for spinlock construction; atomic swap, increment, decrement, and fetch-and-increment can be built around them, failing native support. Hardware memory barriers ensure correct asynchronous communication between parent and helper threads when stopping speculation, as discussed in Section 2.4. Finally, a processor timestamp instruction such as the x86 `rdtsc` instruction can be used for accurate profiling.

3.3 Priority Queueing

Priority queueing allows libspmt to provide some measure of filtering-based control over speculation by assigning thread priorities. Priorities are computed by speculation probes using dynamic execution profiles; our eventual intention is to incorporate Whaley's recommendations [37]. A parent thread enqueues a child task on a global priority queue associated with the runtime when `spmt_parent_request_fork()` is called; meanwhile, helper threads busy wait and compete to dequeue children. A helper thread that succeeds in dequeuing a child calls `spmt_host_child_execute()` to perform speculation. We use a simple bounded-height priority queue protected by a single spinlock; the queue itself is a simple array of generic linked list objects, one list per priority. This design is refactored from our previous implementation [29] and follows Shavit's recommendations [32].

3.4 Single Threaded Mode

We previously described a complex single-threaded mode as part of our work on SableSpMT [28]. This mode operated by saving the parent state at a fork point and switching to child execution within the same operating system thread. It provided critical support for system debugging by making race conditions between parents and children deterministic. Based on our refactoring, we established a *virtual* single-threaded speculation mode in libspmt, as shown in Figure 5. The re-engineered implementation is much simpler, entirely hidden from the host, and the control flow is nearly identical to that of normal speculative execution. As before, more than one parent can execute simultaneously.

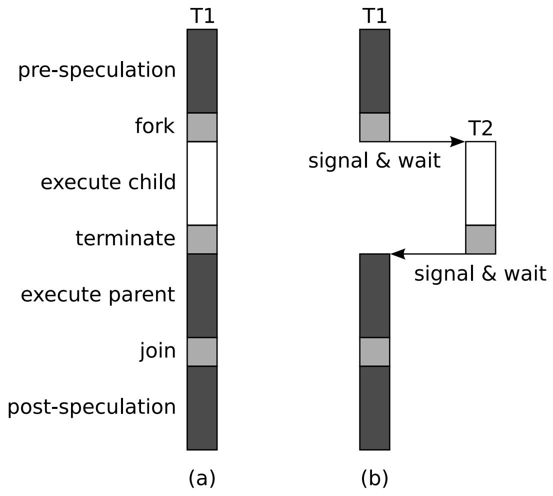

Figure 5: *Single-threaded modes.* (a) The single-threaded mode as originally implemented in SableSpMT. A single thread T1 executes both non-speculative and speculative code. (b) The *virtual* single-threaded mode as implemented in libspmt. The parent thread T1 blocks while the helper thread T2 executes the child.

A boolean runtime option is the only control available to a host. Instead of using the same parent thread to execute the child, the parent will block until the child has completed execution. The rest of execution remains unchanged, and in particular the priority queue is still used. The mode does have somewhat different execution semantics now: instead of interleaving non-speculative and speculative execution in a single thread, either a given parent thread is executing, or one of its children is being executed by a helper thread.

3.5 Dependence Buffering

The dependence buffer uses read and write sub-buffer hashtables; a write occurs directly to the write buffer, whereas a read searches the write buffer, and then the read buffer, and then main memory in order to load a value. Validation checks all reads against main memory, and committal flushes all writes; these operations are managed by libspmt parent objects at child join time.

The dependence buffer is now implemented as a simple wrapper around an open addressing hashtable module used for the read and write sub-buffers. The same table module is also used for the speculation probe registries and hashtable-based predictors. Our previous dependence buffer model operated equivalently [29], but the implementation was complicated and difficult to modify. Instead of accepting a size parameter and masking reads and writes appropriately, there was a macro-generated version for each Java primitive type. Additionally, the hashtable code itself was not usable by any other modules.

3.6 Value Predictors

Value prediction is used by speculative systems to provide guesses for values of heap and static memory locations, automatics, and return values. Importantly, return value prediction allows speculative children to progress beyond consumption of a return value without a violation. A set of value predictors is used for each speculation probe whose associated continuation consumes a return value. The final predicted return value is passed to `spmt_host_child_execute()`, and predictors are updated with actual return values at speculation join points.

Each value predictor is now a separate and minimally sized module that operates on 64-bit values, whereas previously there was considerable shared state, specialization according Java primitive types, and a predictor-specific hashtable implementation, all premature and even potentially harmful optimizations. The net result is that now predictors can be used independently of each other, enabling a key set of optimizations that focus on reducing predictor overhead by adaptively disabling predictors at runtime. The other advantages of independent predictor modules are that: 1) unit testing is actually feasible; 2) it is easier to reuse them in different contexts, for example load value prediction; 3) it is now trivial to experiment with new predictor designs.

The set of available predictors is unchanged from our initial implementation [27]. These include a last value predictor that simply predicts the last value, a stride predictor that captures a constant difference between values, a two-delta stride predictor that functions like the stride predictor but updates the stride it uses after two identical strides in a row, a parameter stride predictor that captures a constant difference between the return value and one input parameter, a context predictor that hashes together a value history to lookup a value in a hashtable, a memoization predictor that hashes together input arguments to lookup a value in a hashtable, and a hybrid predictor that selects the best sub-predictor over the last N values.

3.7 Memory Management

libspmt provides a custom memory management system for child tasks and the objects they contain. `malloc()` and `free()` suffice for the majority of data structures, which are allocated infrequently, but more efficient memory management is necessary for objects that are allocated and freed repeatedly. This is generally not a performance problem for hardware SpMT proposals, as they tend to manage threads through architecture extensions.

Our specific performance issue is a producer-consumer relationship where any thread in the system can allocate, any other thread can free, and each thread runs on a separate processor. We experimented with various standalone `malloc()` replacements including Hoard [2] and TCMalloc [15], but there were several problems. Both depend on libstdc++, a large software component that introduces another dependency in and of itself, lending to build complexity. We also found that successful compilation depended on particular kernel and system library versions, and we wanted libspmt to remain as generic and portable as possible. Finally, we could not find a `malloc()` replacement that targeted our exact problem. However, once the problem was well-defined, implementing a solution was comparatively easy.

Previously, SableSpMT maintained a free list of children per parent thread; it is too expensive for a parent thread to call `malloc()` every time it forks a child, and `free()` every time it joins one. However, this meant that helper threads could not allocate or free children themselves, which had two fairly significant consequences. First, it ruled out nested speculation, which depends on children forking and joining new children independently of their parent. Second, helper threads could not cleanly be used for decoupled and expensive finalization of child tasks after joining, as some parent must still perform an eventual `free()`.

Our solution is similar to that employed by Hoard. Hoard maintains per-processor heaps, and migrates memory to and from a global heap as appropriate. However, threads sharing a processor must lock the processor heap even in the absence of contention. We can simply employ per-thread free lists, because there is a one-to-one mapping between threads and processors. Furthermore, Hoard is a general memory manager, whereas our solution manages only a single child type.

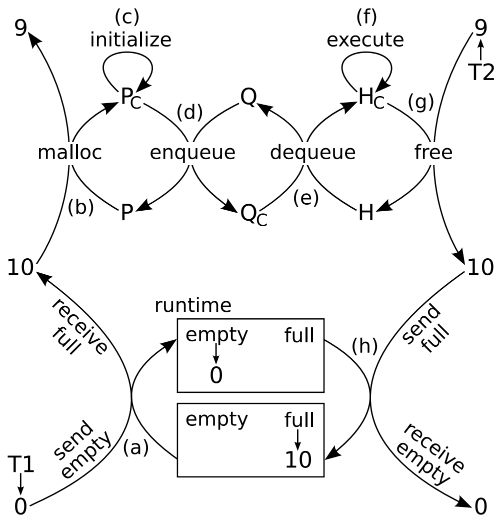

Figure 6: *Producer-consumer memory management.* Initially, parent P requests a new child from its thread T1, but T1 only has an empty block of children on its block list. **(a)** T1 acquires a global runtime lock and exchanges the empty block for a full one with 10 children. **(b)** T1 then allocates a child from this block, reducing the number of children in the block to 9, and returning the child to the parent P, which becomes P_C. **(c)** P_C initializes the child and **(d)** enqueues it on the priority queue Q, which becomes Q_C whilst P_C returns to its P form. **(e)** At some later point, helper H associated with thread T2 dequeues the child, the queue returning to Q form and H becoming H_C. **(f)** H_C now executes the child, and **(g)** when finished frees it to T2, becoming H again. T2 initially has a block with 9 children, which becomes a full block of 10 with the free from H_C. **(h)** T2 now acquires a global lock and returns this full block to the runtime in exchange for an empty one. Although this figure illustrates memory recycling, if the runtime is unable to provide a recycled full block to any thread it will create a new one.

Figure 6 illustrates a dynamic instance of the producer-consumer memory management in libspmt. Migration between threads and the runtime occurs using blocks, which contain some constant number of children, the default being 10. The tradeoff between synchronization overhead and excess memory consumption can be controlled by adjusting the block size.

When a thread tries to allocate a child, it checks for a non-empty block on its free list. If it finds one, it simply removes and returns a child without any synchronization. If it does not find one, it exchanges an empty block for a full one with the runtime, using global synchronization. Upon freeing a child, if the thread creates a full block, and the number of full blocks on the thread free list exceeds some threshold, that block is returned to the runtime in exchange for an empty one.

If the runtime is unable to satisfy a request for a full block, it calls `spmt_child_create()` once for each element in a block. After allocating a dependence buffer and stack mirror, this in turn will use the `spmt_host_child_create()` callback to create host execution environments. Upon thread shutdown all blocks are returned to the runtime, and upon runtime shutdown, `spmt_child_destroy()` is called for each child in each block, which in turn uses the corresponding `spmt_host_child_destroy()` callback.

For other frequently required types that are not involved in a producer-consumer relationship, such as hashtables that are freed and allocated upon expansion, simple global or per-thread free lists suffice.

4 Development

Development of any kind of independent library is a challenge in software engineering, and the complexity can easily result in a difficult development process. In order to accelerate our efforts and build on previous experience, we worked from our initial SableSpMT implementation, and did not simply discard it. We followed numerous software design principles, as discussed in the remainder of this section: refactoring, modularity, client or host independence, portability at both hardware and operating system levels, unit testing and test-driven development, regular and automated builds, and heavy exploitation of C compiler features.

Refactoring. The advice given in *Refactoring* [12], *Test-Driven Development* [1], and *Code Complete* [24] was applied where possible. These are three well-known books in software engineering, and provide recipes for developing and maintaining flexible code bases. At all times during the development of libspmt the focus was on keeping SableSpMT able to run the SPECjvm98 benchmark suite.

Language Independence. The library is independent of Java and language independent in general. Previously, our implementation of SableSpMT was heavily tied to Java Native Interface (JNI) and VM types, and included a fair amount of macro-based specialization. A language with a call stack and source programs with relatively frequent function calls are required, due to the implementation of speculative method level parallelism. However, individual modules such as the value predictors and dependence buffer can also be reused and replaced independently without any need for SMLP.

Portability. libspmt does have several build dependencies. GNU Autoconf, Automake, m4, and Libtool are used as a highly portable build system. POSIX pthreads are used for multithreading. The source code is ISO C90, although presently GCC is necessary in order to handle small amounts of inline assembly for atomic operations, as well as 64-bit `long long` types, but neither of these things are a major barrier to compilation with other compilers. It compiles with as many lint-like GCC warning flags enabled as reasonable.

Modularity. The library design is object-oriented and generally divided into modules or classes. Containment is used as an alternative to an inheritance scheme based on function pointers. There are no global variables, only global constants. Functions are small and performance depends heavily on a good C inliner. The use of the C preprocessor is avoided as much as possible. There is one `struct` defined per `.c` file, and this `struct` is opaque, and accessible only through the corresponding `.h` file `typedef` and functions that operate on it. Each module has functions with private, library-wide, and public visibility. The package prefix is `spmt`, and this appears before all symbols except local variables, both private and public, so as to simplify the process of exposing and hiding symbols.

Unit Testing. Each module is independently unit tested using the Check [23] unit testing framework. Those modules written from scratch were developed using test-driven development [1], and thus have comprehensive unit tests, whereas modules migrated from SableSpMT are not as comprehensively tested. However, the presence of the framework makes debugging and future test writing easier, and allows for bug fixes to be driven by regression tests.

Compilation. libspmt is compiled independently of its hosts. This can permit the use of a more aggressive set of compiler optimization flags. For example, SableSpMT cannot safely be compiled with global common subexpression elimination optimizations due to particular GCC limitations with respect to the use of labels as values, whereas libspmt can; thus code that was once tied up in SableSpMT can now be more heavily optimized.

The build system also supports two broadly different compilation modes. The first is the traditional method for compiling C programs, where each module is compiled independently, and then all object files are linked together. The second `#include`'s all `.c` files into a giant source file, and compiles that to a single object file. The advantages of the first mode are that it allows for proper unit testing, enforces modularity, and simplifies debugging. The advantage of the second mode is that multiple translation units are compiled at once, enabling whole program optimization, similar to how the `-qipa` XL C and `-combine -fwhole-program` GCC flags operate.

`./configure` options are also available for controlling other wide-sweeping changes. These include enabling debugging, assertions, statistics gathering, profiling, and aggressive compiler optimization.

5 Related Work

Various software systems have been designed to support parallel execution, and include both speculative and non-speculative approaches. The well-known *Cilk* language is based on a sophisticated runtime environment for non-speculative, fine-grain parallelization with automatic load balancing, and is guided by explicit programmer specifications [13]. The *zJava* compiler and runtime system is a more recent and VM-related example. zJava depends on symbolic access paths computed at compile time to parallelize a program dynamically, without using programmer directives [6]. Method calls are executed in separate child threads, while the parent executes the method continuation until either a return value is consumed or a heap data dependence is encountered, at which point it blocks. A registry of running threads, methods, and heap regions is maintained to enforce sequential execution semantics. In general, non-speculative implementations such as these exchange the complexity of speculative execution designs for increased complexity in ensuring correct memory access orderings.

Speculative parallelization designs were originally proposed in the context of hardware, but several software approaches have also been attempted. Both hand-done proofs-of-concept [20] and full implementations have been demonstrated. Papadimitriou and Mowry, describe a software system for thread level speculation based on a virtual memory page protection mechanism [25]. Conflicting memory accesses between threads are caught and memory is synchronized using standard page trapping and signal handling. Other approaches follow hardware designs in tracking individual memory access conflicts. The loop-based speculative system proposed by Cintra and Llanos exploits both compiler analysis and runtime testing to identify shared variables and handle individual dependence violations [8]. Softspec is a compiler and runtime system that parallelizes loops in C programs with *stride-predictable* memory references [4]. A memory reference in a loop is stride-predictable if it changes by a constant value or *stride* with each iteration. Softspec uses compiler-inserted calls to a runtime system that dynamically splits loops into parallel threads with multiple iterations each. An undo log is maintained for rollback, and barrier synchronization is used to join the loop threads.

Hybrid software and hardware designs have of course also been investigated, and in fact most hardware proposals include significant software support in the form of a compiler and runtime system [7, 33]; a general purpose compiler for SpMT hardware architectures has even been proposed [3]. Our library design is based on an understanding of the features common to these designs, the obvious need for a generic interface, and of course our own experience in software SpMT implementation [28, 29].

Garzaran *et al.* proposed a taxonomy for state buffering mechanisms in thread level speculation [14]. According to that taxonomy libspmt supports Eager Architectural Main Memory (Eager AMM), as speculative threads write variables to a dependence buffer and not directly to main memory, and the buffer is committed immediately at join time along with the child stack. It also supports multiple tasks and multiple versions of variables per processor (MultiT&MV): per-processor helper threads begin execution of child tasks as soon as both a helper and child are available, and each child has its own dependence buffer. This design is recommended as the most effective in terms of benefits gained for the complexity of implementation.

Techniques developed for thread level speculation are employed in other contexts as well. Eugster demonstrated a debugging environment for concurrent programs based on the essential idea of *rollback* [11]. Saving state and rolling back execution allows different scheduling choices to be considered in debugging or exhaustively considered in testing. Persistent designs also require basic program state checkpointing to restore the system to a previous, interrupted execution [9, 34]. Concepts such as rollback and checkpointing are of course important in fault-tolerant schemes, allowing correctness to be ensured by saving state and replaying an execution if failure is detected. This also provides the basic mechanism for transactional execution, which can be used to give Java *codelets* ACID properties [30].

Actual transactional approaches are of course quite closely related [19]. Techniques for *software transactional memory* [31] require efficient enforcement of atomicity, and thus many of the same mechanisms as thread level speculation, including isolation of code execution and general safety enforcement. Optimistic designs further share a need for dynamic conflict management, buffering, validation, and other runtime components typical of speculative multithreading designs [36]. Transactional language implementations such as Herlihy *et al.*'s *Dynamic Software Transactional Memory* [18], Harris and Fraser's lightweight transactions [17], and most recently Atomos [5] all rely on the ability to safely execute code and restore or repair state invisibly to the user. As with other speculative multithreading designs, libspmt manages considerable extra complexity in order to provide facilities suitable for automatic parallelisation of single-threaded code, with no requirement for programmer interaction. Supporting transactional execution in a speculative multithreading environment is certainly feasible, however, and represents a potential application of our work.

6 Conclusions & Future Work

Both virtual and non-virtual execution environments are expected to exploit underlying performance-enhancing features. However, supporting advanced hardware and operating system features is not trivial, and complex enhancements like speculative multithreading generally require a tight and highly-specialized integration of the runtime system and speculative components. As a complex optimization a perfect separation is not possible, but a more maintainable approach is clearly desirable if multiple clients are expected to use the same speculative system.

Our libspmt library for method level speculation is a modular approach to the use of thread level speculation. libspmt is designed as a well-defined set of modular components that provides a reasonably minimal interface to its unique runtime system for managing thread level speculation. Host or client services that interact closely with the speculative system are cleanly separated, and allow for relatively arbitrary execution designs to support SpMT. We have paid careful attention to good design concerns, which include both recommended software engineering practices and the requirement that our strong modularity does not degrade performance. Our reference SableSpMT client that is an extension of a Java interpreter demonstrates both the feasibility and the efficiency of our design.

The newly available modularity in libspmt will allow further work to progress in several areas independently. Improving fork heuristics is important [37], as is adaptive value prediction [28]; previous profiling indicated that these will help to address a significant percentage of overhead in our combined SableSpMT–libspmt system. More involved improvements to libspmt consist of allowing speculative threads to spawn speculative children themselves, that is, supporting in-order nested speculation, extending and reusing our value predictors to support efficient prediction of arbitrary load values, and investigating speculative locking and support for transactional language systems. Our immediate work is focused on integrating libspmt into IBM's production Java JIT compiler. JIT code generation is more complex than interpreter code, but permits many useful internal optimizations, among them relatively easy code specialization. Our experience in this work is expected to guide the integration of libspmt into other language environments.

We would like to thank the members of the University of Toronto Compiler and Architecture Reading Group and our colleague Michael Batchelder at McGill University for their preliminary feedback on this material. This research was funded by an IBM CAS fellowship and the Natural Science and Engineering Research Council of Canada.

References

[1] K. Beck. *Test-Driven Development: By Example*. Addison Wesley Professional, 1st edition, Nov. 2002.

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

[3] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 15(8):713–724, Aug. 2004.

[4] D. Bruening, S. Devabhaktuni, and S. Amarasinghe. Softspec: Software-based speculative parallelism. In *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, Dec. 2000.

[5] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–13, June 2006.

[6] B. Chan and T. S. Abdelrahman. Run-time support for the automatic parallelization of Java programs. *Journal of Supercomputing*, 28(1):91–117, Apr. 2004.

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

[8] M. Cintra and D. R. Llanos. Design space exploration of a software speculative parallelization scheme. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 16(6):562–576, June 2005.

[9] A. Cunei and J. Vitek. A new approach to real-time checkpointing. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE)*, pages 68–77, June 2006.

[10] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *OOPSLA'03: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 149–168, Oct. 2003.

[11] P. Eugster. Java virtual machine with rollback procedure allowing systematic and exhaustive testing of multi-threaded Java programs. Master's thesis, ETH Zürich, Zürich, Switzerland, Mar. 2003.

[12] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1st edition, June 1999.

[13] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, June 1998.

[14] 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.

[15] S. Ghemawat and P. Menage. TCMalloc: Thread-caching malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[16] B. Goetz. Optimistic thread concurrency: Breaking the scale barrier. Technical Report AWP-011-010, Azul Systems, Inc., Mountain View, CA, USA, Jan. 2006.

[17] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA'03: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 388–402, Oct. 2003.

[18] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, July 2003.

[19] T. A. Johnson, S.-I. Lee, S.-J. Min, and R. Eigenmann. Can transactions enhance parallel programs? In *Proceedings of the 19th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Nov. 2006. To appear.

[20] I. H. Kazi and D. J. Lilja. JavaSpMT: A speculative thread pipelining parallelization model for Java programs. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 559–564, May 2000.

[21] V. Krishnan and J. Torrellas. A direct-execution framework for fast and accurate simulation of superscalar processors. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 286–293, Oct. 1998.

[22] P. S. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing (ISPP)*, pages 165–171, Jan. 1994.

[23] A. Malec, S. Neumann, F. Hugosson, and C. J. F. Pickett. Check: A unit testing framework for C. `http://check.sourceforge.net/`.

[24] S. McConnell. *Code Complete*. Microsoft Press, 2nd edition, June 2004.

[25] S. Papadimitriou and T. C. Mowry. Exploring thread-level speculation in software: The effects of memory access tracking granularity. Technical Report CMU-CS-01-145, School of Computer Science, Carnegie Mellon University, Pittsburg, PA, USA, July 2001.

[26] C. J. F. Pickett and C. Verbrugge. Compiler analyses for improved return value prediction. Technical Report SABLE-TR-2004-6, Sable Research Group, School of Computer Science, McGill University, Montréal, Québec, Canada, Oct. 2004.

[27] 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.

[28] 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.

[29] 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.

[30] A. Rudys and D. S. Wallach. Transactional rollback for language-based systems. In *Proceedings of the 3rd International Conference on Dependable Systems and Networks (DSN)*, pages 439–448, June 2002.

[31] N. Shavit and D. Touitou. Software transactional memory. In *PODC'95: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, Aug. 1995.

[32] N. Shavit and A. Zemach. Scalable concurrent priority queue algorithms. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 113–122, May 1999.

[33] 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.

[34] S. J. Tjasink. *PLaVa: A Persistent, Lightweight Java Virtual Machine*. PhD thesis, Department of Computer Science, University of Cape Town, Rondebosch, South Africa, Feb. 1999.

[35] R. Vallée-Rai. Soot: A Java bytecode optimization framework. Master's thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, July 2000.

[36] 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.

[37] J. Whaley and C. Kozyrakis. Heuristics for profile-driven method-level speculative parallelization. In *Proceedings of the 34th International Conference on Parallel Processing (ICPP)*, pages 147–156, June 2005.