SableSpMT: A Software Framework for Analysing Speculative Multithreading in Java

Christopher J.F. Pickett Clark Verbrugge

School of Computer Science, McGill University
Montréal, Québec, Canada H3A 2A7
{cpicke,clump}@sable.mcgill.ca

Speculative multithreading (SpMT) is a promising optimisation technique for achieving faster execution of sequential programs on multiprocessor hardware. Analysis of and data acquisition from such systems is however difficult and complex, and is typically limited to a specific hardware design and simulation environment. We have implemented a flexible, software-based speculative multithreading architecture within the context of a full-featured Java virtual machine. We consider the entire Java language and provide a complete set of support features for speculative execution, including return value prediction. Using our system we are able to generate extensive dynamic analysis information, analyse the effects of runtime feedback, and determine the impact of incorporating static, offline information. Our approach allows for accurate analysis of Java SpMT on existing, commodity multiprocessor hardware, and provides a vehicle for further experimentation with speculative approaches and optimisations.

Keywords Java, virtual machines, speculative multithreading, thread level speculation, profiling, static and dynamic analysis.

1. Introduction

Speculative multithreading (SpMT), also known as thread level speculation (TLS), is a promising technique for dynamic parallelisation of sequential programs. It has been investigated through many hardware proposals and simulations [2, 4, 5, 8, 11, 14, 16, 17, 20, 21, 24, 25, 30, 31, 34, 35, 36, 37, 39, 41], and a smaller but not insignificant number of software designs [3, 6, 18, 19, 26, 27, 28, 32, 40, 42], each offering its own analysis of various implementation and optimisation techniques. However, it is difficult to evaluate these proposals with respect to and in combination with each other, as there are multiple source languages, thread partitioning schemes, SpMT compilers, and hardware simulators being used. Even if these variables remain fixed, it is highly unlikely that an identical software architecture and/or set of simulation parameters will be used.

We present SableSpMT as a common framework and solution to these problems, as an extension of the SableVM Java virtual machine [13]. SableSpMT provides a convenient hardware abstraction layer by operating at the bytecode instruction level, takes the full Java language and VM specification into account, supports static analysis through the Soot bytecode compiler framework [38] and parsing of Java classfile attributes [29], and works on existing multiprocessor systems. SableSpMT provides a full set of SpMT support features, including generic *speculative method level parallelism* (SMLP) and *return value prediction* (RVP). Our work is designed to facilitate SpMT research, and includes a unique debugging mode, logging, and portability amongst the features that make it appropriate for experimentation and new designs.

We report on both Java benchmark and framework behaviour to illustrate the forms of experimental and design analysis we support. Through dynamic measurements we show that while *speculative coverage*, the percentage of sequential program execution that occurs successfully in parallel, can be quite high in Java programs, the overhead costs are significant enough in our initial implementation to preclude actual speedup. However, we are able to perform experiments to determine upper bounds on speedup in the absence of all overhead, and our execution times are still better than those offered by hardware simulators providing similar functionality [20]. We also break down the SpMT overhead costs to determine performance bottlenecks and set optimisation goals. In our case overhead is dominated by verification of speculative threads and the concomitant interprocessor memory traffic, lock and barrier synchronization, and update costs for return value prediction (RVP), a runtime optimisation technique that can improve SMLP performance by up to 2-fold [16]. With regards to RVP, results gathered within our framework extend previous studies to include more realistic benchmark runs, offer further data on the relative benefits, requirements and costs of various prediction strategies, and expose the potential benefits of exploiting both static and runtime feedback optimisation information.

Hardware simulations have already demonstrated the great potential in speculative multithreading. We contend that the same techniques, however, can be investigated more generally and efficiently at the virtual machine level using commodity multiprocessor hardware, given an appropriate analysis framework. Virtual machines allow for exploration of complex design changes, facilitate detailed instrumentation, provide high level information that is not generally available to hardware approaches, and are able to interact directly with the underlying architecture. Our work is intended to enable SpMT investigations by providing an execution and analysis environment, a general design and componentry, and real data from a working implementation.

1.1 Contributions

We make the following specific contributions:

- We describe SableSpMT, a complete implementation of SpMT for Java that runs on real multiprocessor hardware, and present its suitability as an analysis framework. This is the first complete such work within a virtual machine.

PASTE'05 September 5–6, 2005, Lisbon, Portugal.

- We simplify the implementation and analysis of new SpMT designs by providing a deterministic, single-threaded uniprocessor mode, as well as logging facilities, statistics gathering, and full JVM support.
- We demonstrate that high level analysis information can be easily exploited by our framework. Ahead-of-time results computed by Soot as well as runtime profiling-based feedback can passed to our execution engine to improve performance, and we illustrate the technique using our work on RVP.
- We provide detailed data on the speculative execution of non-trivial programs, which include a breakdown of overhead costs, the impact of highly accurate RVP, two different measurements of dynamic parallelism, and overall running times.

In Section 2 we discuss related work on the analysis of speculative multithreading. In Section 3 we describe the general SpMT model we have used and give an overview of how our framework is constructed and its main features. This includes an exposition of the components required for Java SpMT, our multithreaded execution and single-threaded debugging modes, system configuration options, and the data logging and trace generation features. In Section 4 we analyse actual data and show the flexibility of our system in terms of data gathering. Finally, we discuss future work and conclude in Section 5.

2. Related Work

Speculative multithreading approaches have been developed primarily in the context of novel hardware environments. A number of general purpose speculative architectures such as the Multiscalar architecture [11], the Superthreaded architecture [37], MAJC [36], Hydra [14], and several other designs [21, 34] have been proposed, and simulation studies have generally shown quite good potential speedups. Steffan *et al.* give a recent implementation and good overview of the state of the art in [35].

From the speculative hardware level, an executing Java virtual machine does not have distinguished performance in comparison with other applications [39]. As an interpreted language, however, Java can provide higher level abstractions and information than generic machine code. High level program information is used in a few hybrid software/hardware studies, including Chen and Olukotun's thread level speculation system for Java [5]. Java traces applied to simulated architectures have been used by several researchers, including Hu *et al.* in their study of the impact of return value prediction [16], and Whaley and Kozyrakis' recent study of heuristics for method level speculation [41].

Software architectures for SpMT are less common. Rundberg and Stenström describe a software approach to speculation in C [32]. Their prototype implementation shows good speedup, but is verified only through hand done transformations and greatly limited real world testing. Kazi and Lilja describe a software library for *coarse-grained thread pipelining* [19], demonstrated through manual parallelisation of loops in C programs. The "Softspec" software speculation environment [3] concentrates purely on loop-based speculation. The approach depends on machine code level runtime profiling to identify independent loop bodies suitable for speculative execution. Cintra and Llanos have developed a FORTRAN-based system that also speculates on loop bodies [6]. These approaches all achieve good performance results, but none are based on Java or designed specifically as experimental frameworks.

Only very limited studies on language level speculation for Java have been done previously. Yoshizoe *et al.* give results from a partially hand-done loop level speculation strategy implemented in a rudimentary (e.g., no GC) prototype VM [42]. They show good speedup for simple situations, but lack of heap analysis limits their results. A more convincing analysis is given by Kazi and Lilja through manual Java source transformations [18]; similarly Welc *et al.* demonstrate good speedup of loops in easily parallelisable benchmarks, through application of *safe futures* for Java [40], source level annotations that provide SpMT-like functionality and depend on VM support for parallelisation. Opposingly, Warg and Stenström argue that Java-based SpMT has inherently high overhead costs which can only be addressed through hardware support [39]. Our data and analysis are significantly more comprehensive than prior studies, and suggest that while overheads can be quite high, there is sufficient potential parallelism to offset the cost.

We have analysed return value prediction as part of our investigation into SpMT behaviour. Value prediction for SpMT has been explored by several groups [4, 8, 25], and is generally well-studied. The specific utility of return value prediction for method level speculation in Java was shown by Hu *et al.* [16], with further prediction accuracy investigated by the authors [26, 27]; our value prediction approach here is based on these designs.

3. Framework

We begin with an overview of our framework, followed by a brief exposition of our speculative execution model, and some of the features of our framework that help with implementation and debugging of such a complex undertaking.

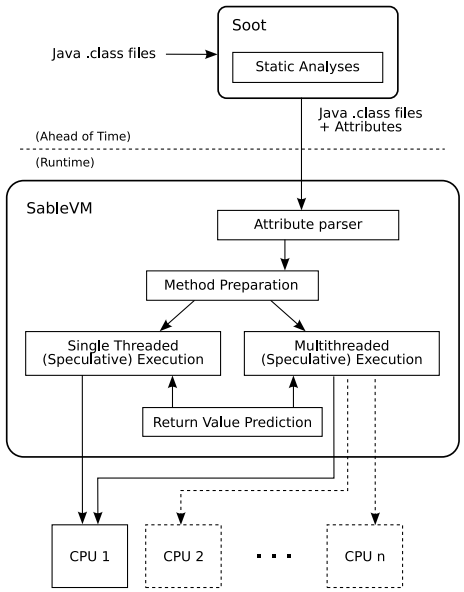

Figure 1. *The overall SableSpMT execution environment.*

The overall SableSpMT execution environment is shown in Figure 1. SableVM prepares special speculative and normal non-speculative versions of methods at runtime from dynamically loaded classes, which are read in from Java `.class` files. Soot [38] is used to transform, analyse, and attach attributes to these classes in an ahead-of-time step [29], although this could also occur at runtime. Two execution modes are provided, a single-threaded "simulation" mode and a true multithreaded mode, both of which can exploit a return value prediction framework. The multithreaded mode splits single Java threads across multiple processors on an SMP machine.

3.1 Speculative Method Level Parallelism

Threads are created using the SpMT variant known as *speculative method level parallelism* (SMLP) [4, 5, 16, 39, 41], depicted in Figure 2. Ordinary sequential bytecode execution is shown in

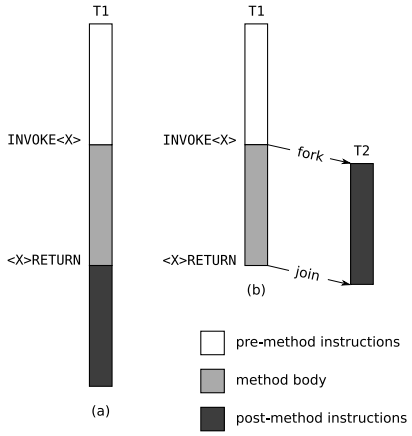

Figure 2. *Sequential and SMLP-based bytecode execution.*

Figure 2a, where the target method of an `INVOKE<X>` instruction executes before the instructions following the return point. Speculative SMLP-based execution is shown in Figure 2b. Upon reaching a method callsite, the non-speculative parent thread `T1` forks a speculative child thread `T2`. If the method is non-void, a predicted return value is pushed on `T2`'s operand stack. `T2` then continues past the return point in parallel with the execution of the method body, buffering all reads from main memory. When `T1` returns from the call, it joins `T2`. If the actual return value matches the predicted return value, and there are no dependence violations between buffered reads and post-invoke values, `T2`'s buffered writes are committed and non-speculative execution jumps ahead to where `T2` left off, yielding speedup. If there *are* dependence violations or the prediction is incorrect, `T2` is simply aborted.

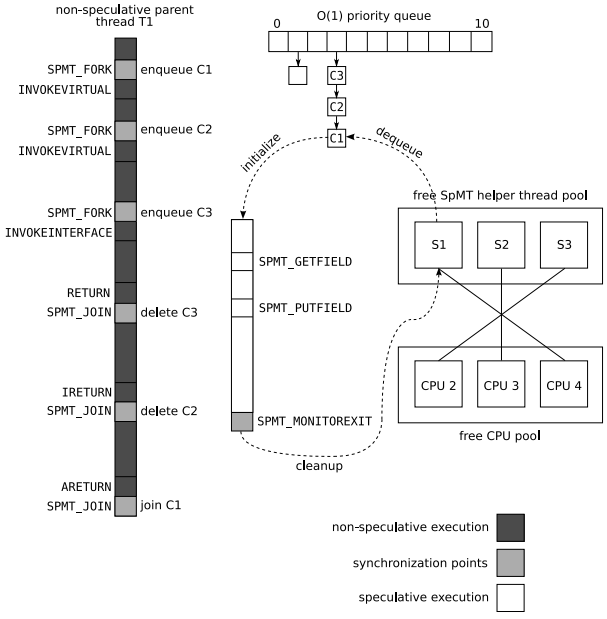

Figure 3. *Multithreaded mode.*

Many components are needed for SMLP to work properly in a JVM, the full details of which are given in [28]. A high level view of the multithreaded mode that brings together all of these components is shown in Figure 3. A *dependence buffer* protects main memory from out-of-order and possibly invalid speculative operations, and some form of *stack buffering* is necessary to give child threads a protected execution context. New `SPMT_FORK` and `SPMT_JOIN` instructions surround every callsite; the fork instruction inserts child threads into a priority queue, which are dequeued and executed on separate processors by SpMT *helper threads*, and the join instruction stops and validates children, either committing or aborting them. While executing speculative code, we need *modified bytecode instructions* to protect against unsafe control flow; for example, `GETFIELD` is modified to read from a dependence buffer, and `MONITOREXIT` causes speculation to come to an abrupt halt, although it does not automatically force abortion. Finally, we need to make sure speculation interacts safely with exception handling, garbage collection, native method execution, synchronization, class loading, and the new Java memory model [23].

We make several different optimisations to these components in SableSpMT, some of the more notable ones being aggressive return value prediction [26, 27], improvements to the dependence buffer, allowing for speculative threads to enter and exit methods, better enqueuing algorithms, speculative object allocation, and reduction of interprocessor memory traffic. Most of the techniques we have encountered in the literature can also be implemented within our framework; in Section 4 we illustrate typical data gathering and analysis using our work on return value prediction and the speculative engine itself as examples.

3.2 Single-threaded Mode

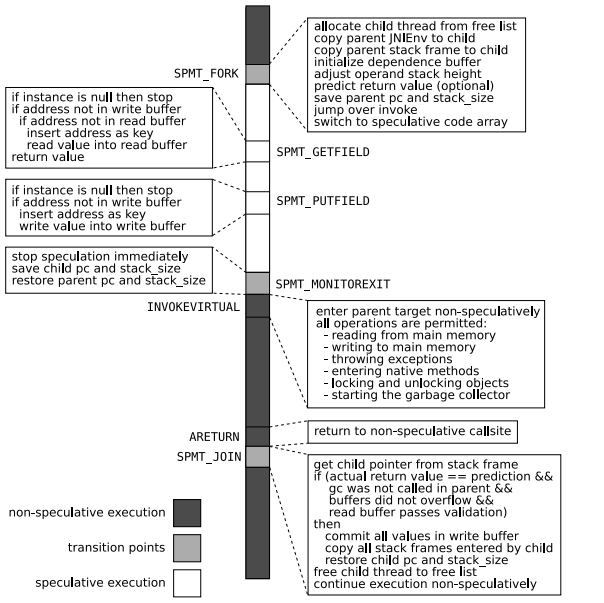

Figure 4. *Single-threaded simulation mode.*

One of the unique features of our design is a single-threaded simulation mode that mimics the process of speculative execution in a single thread. Early on in the development of SableSpMT, we found ourselves wanting some way to test the components we had written in the context of an executing JVM, without introducing the complexity of actual concurrency into our debugging process. The resulting deterministic design is shown in Figure 4. In this mode a single thread of Java execution follows the complete speculative control flow. Upon reaching a fork point, the method call is skipped, and the ensuing code is executed speculatively; when a terminating condition is reached, the same thread jumps back to the non-speculative execution of the method call, and upon returning from the call, it attempts to join with its own speculative result.

There are three primary advantages to having this single-threaded simulation mode. First, it allows for testing of SpMT components in an incomplete system, most importantly one without multiprocessor support. It does so by providing state saving and restoral, and interleaving the execution of speculative and non-speculative code. Second, by not running multiple threads it prevents race conditions, deadlocks, and memory traffic from interfering with development, helping to minimize the search space when faced with debugging. We were able to alternate coding with designing support for SpMT according to the full JVM Specification [22], and only after we had completed a requirements analysis in this manner did we develop the multithreaded execution mode. Third, it means we have the foundations for Java checkpointing and rollback within a virtual machine. This has utility for Java outside of SpMT, in traditional debugging [7], database transactions (e.g. in `java.sql.Connection()`), formal verification [10], fault-tolerance [12], and software transactional memory [15].

3.3 System Configuration

System properties specified on the command line are used to select different SpMT algorithms and data structures, which facilitates experimental analysis by eliminating the need for multiple VM builds. As changes to SableSpMT are introduced, rather than outright replace old control flow or adjust constants to optimal values, system properties are used wherever possible, and thus it is straightforward to make controlled comparisons and revert to old configurations. In finalized builds, these properties can be automatically converted to constants via preprocessor directives and a single Autoconf `configure` option, so that the added runtime overhead of conditionals testing them will be optimised away. At the time of writing, there are over 50 such properties, controlling everything from maximum RVP hashtable sizes to the number of executing SpMT helper threads, and it is easy to introduce new ones. The only other significant compile-time `configure` options in SableSpMT allow the user to 1) enable SpMT in the first place, 2) enable debugging and assertions, and 3) enable statistics gathering for post-execution analysis.

3.4 Logging and Trace Generation

```
T1:P16384 - @0x2a976bad68 ALOAD_0
T1:P16384 - @0x2a976bad70 SPMT_FORK
T1:P16384 - ENQUEUE SPMT CHILD @0x5ed850
T1:P16384 - @0x2a976bad88 INVOKESPECIAL
T1:P49156 S DEQUEUE SPMT CHILD @0x5ed850
T1:P49156 S START SPMT
T1:P16384 - entering java/lang/Object.<init>()V
T1:P49156 S @0x2a976baf38 ALOAD_0
T1:P16384 - @0x2a976baf90 RETURN
T1:P49156 S @0x2a976baf40 ALOAD_1
T1:P16384 - exiting  java/lang/Object.<init>()V
T1:P16384 - @0x2a976badb0 SPMT_JOIN
T1:P49156 S @0x2a976baf48 SPMT_PUTFIELD
T1:P16384 - signalling spmt thread halt @0x5ed850
T1:P49156 S STOP SPMT - SIGNALLED_BY_PARENT
T1:P16384 - SPMT PASSED @0x5ed850
```

Figure 5. *SpMT execution trace.*

Finally, SableSpMT provides a comprehensive logging and trace generation system that can present Java SpMT events by themselves, or interleave them with existing execution traces of class loading, method invocation, garbage collection, synchronization, and bytecode execution. An example trace with interleaved method invocation, bytecode, and SpMT events is shown in Figure 5. These traces are primarily useful for debugging purposes when implementing new techniques. SableVM supports only the JVMDI and JDWP for integration with debuggers at this time, and although we do not provide trace compression or an implementation and extension of the related JVMPI or JVMTI profiling interfaces, these facilities could be incorporated to permit detailed analysis of SpMT execution traces, using a dynamic metrics tool such as *J [9].

4. Experimental Analysis

In this section we describe our return value prediction framework, the integration of static analyses into SableSpMT, and various kinds of dynamic analysis available to the researcher. We provide experimental results to demonstrate how these analyses give insight into the runtime behaviour of individual benchmarks, components of the framework, and the framework as a whole, and how they suggest interesting areas for future investigation and optimisation research.

4.1 Return Value Prediction

Method call return values are often used relatively soon after a method call. Return value prediction (RVP) allows a speculating thread to proceed further without failing due to such a dependency by heuristically predicting the return value. It was previously shown that accurate RVP is critical for Java SMLP performance [16], and on this basis we set out to explore highly accurate RVP in SableSpMT [27]. We implemented several well-known predictors, including a context predictor that uses a per-callsite hashed history of the last five return values, and a hybrid predictor that selects the best of several sub-predictors at runtime. We also introduced a powerful new memoization predictor that associates return values with hashed method arguments. When existing predictors were combined in a hybrid we achieved an average accuracy of 72% over SPECjvm98, and the inclusion of the memoization predictor increased this average to 81%. Exploiting VM level knowledge about the width of primitive types allowed us to reduce hashtable memory by 35%.

Full details on the analyses performed in our initial RVP study are available in [27]; two of the neater results obtained were how the context and memoization predictors exhibited dramatically different accuracy depending on benchmark, and how we were able to identify a small percentage of callsites as being responsible for either the production or consumption of highly variable data, according to final context or memoization predictor sizes respectively. The RVP system is now a key part of SableSpMT; it is easily extendable to support new types of return value predictors, and could even be used for general purpose load value prediction by speculative threads.

4.2 Static Analysis

The Soot bytecode compiler framework [38] is a convenient tool for ahead-of-time static analysis and transformation in the absence of the runtime static analysis support typically found in JIT compilers. In Figure 6 we show the use of Soot to transform the base input Java classfiles in order to insert `SPMT_FORK` and `SPMT_JOIN` instructions. The same process can also be used to append static analysis information as classfile attributes [29], which are then interpreted by the SpMT engine.

We use side effect and callgraph information derived from Soot's points-to analysis in two compiler analyses for improved RVP, and study the effect on runtime predictor behaviour using our framework [26]. The first analysis is a *return value use* analysis (RVU), that determines how return values are used after returning from a method call. We find statically that an average 10% of non-void callsites generate *unconsumed* return values, and 21% of callsites generate *inaccurate* return values, which we define as

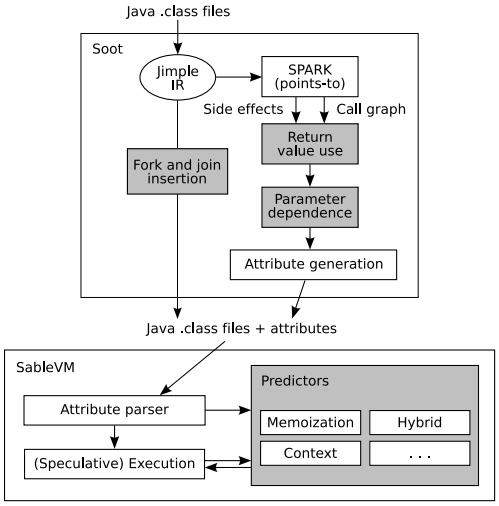

Figure 6. *Static analysis integration.*

those that are used only inside boolean or branch expressions. Actual runtime measurements show less improvement: only 3% of dynamic method invocations return unconsumed values, whereas 14% return inaccurate values. This analysis does reduce hashtable collisions, saving 3% of predictor memory and increasing accuracy by up to 7%.

The second analysis computes *parameter dependence* (PD), a form of slicing that determines which parameters affect the return value. Statically, we observe that 25% of consumed callsites with one or more parameters have zero parameter dependences, and 23% have partial dependences, such that the return value does not depend on one or more parameters. At runtime, however, we find that 7% of dynamic method invocations have zero dependences and only 3% have partial dependences. The results of this analysis are exploited to eliminate inputs to the memoization predictor, and the accuracy of memoization alone for `jack`, `javac`, and `jess` increases by up to 13%, with overall memory requirements being reduced by a further 2%. Although these analyses yield only incremental improvements, at least in their current form, they do demonstrate how new static analyses can be easily incorporated into SableSpMT and both validated and employed at runtime.

4.3 Dynamic Analysis

We now describe dynamic analysis of the SableSpMT engine, an extension of the SableVM 1.1.9 switch interpreter. All experiments were performed on a 1.8 GHz 4-way SMP AMD Opteron machine running Linux 2.6.7, using native 64-bit binaries and running the SPECjvm98 benchmark suite [33] at size 100 (S100). Children were forked at every callsite reached non-speculatively, all free processors were occupied by speculative helper threads, and an optimal return value prediction configuration was used, unless otherwise stated.

4.3.1 Speculation Overhead

The overhead of thread operations in any SpMT system is a major concern [39], and this is especially true in a pure software environment. We introduced profiling support into our framework in order to provide a complete breakdown of SpMT overhead incurred by both parent and speculative child threads; refer to Tables 1 and 2 respectively. As shown in Figure 7, parent threads suffer overhead when forking, enqueuing, joining, and validating child threads, and child threads suffer on startup and when they reach some stopping condition.

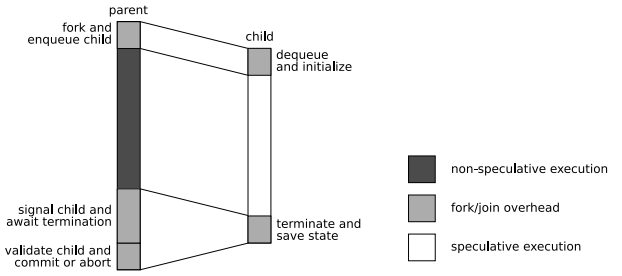

Figure 7. *Speculation overhead.* Both non-speculative parent and speculative child threads suffer wasted cycles due to overhead at fork at join points.

parent execution	comp	db	jack	javac	jess	mpeg	mtrt	rt
USEFUL WORK	39%	24%	29%	30%	21%	59%	49%	58%
initialize child	2%	5%	3%	4%	4%	2%	1%	2%
enqueue child	4%	10%	10%	9%	7%	3%	2%	2%
TOTAL FORK	6%	15%	13%	13%	11%	5%	3%	4%
update predictor	7%	13%	12%	11%	12%	6%	7%	7%
delete child	5%	5%	5%	4%	5%	2%	2%	2%
signal and wait	15%	14%	11%	11%	19%	8%	26%	11%
validate prediction	4%	4%	4%	5%	7%	3%	2%	3%
validate buffer	4%	6%	6%	5%	5%	3%	1%	2%
commit child	5%	5%	7%	6%	6%	3%	2%	3%
abort child	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%
clean up child	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%
profiling	11%	10%	10%	12%	11%	7%	5%	6%
TOTAL JOIN	53%	59%	57%	56%	67%	34%	47%	36%
PROFILING	2%	2%	1%	1%	1%	2%	1%	2%

Table 1. *Non-speculative thread overhead breakdown.* Parent execution consists of useful work, fork overhead, and join overhead, and also the profiling overhead inherent in delineating these three broad tasks. Profiling in the join process includes the cost of gathering overhead info for the other eight sub-tasks, and of updating various SpMT statistics.

helper execution	comp	db	jack	javac	jess	mpeg	mtrt	rt
IDLE	86%	82%	78%	78%	78%	55%	53%	71%
INITIALIZE CHILD	3%	4%	4%	4%	4%	2%	5%	4%
startup	<1%	<1%	<1%	<1%	<1%	<1%	1%	<1%
query predictor	3%	5%	4%	4%	6%	5%	15%	8%
useful work	5%	6%	10%	10%	10%	34%	20%	13%
shutdown	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%
profiling	<1%	<1%	<1%	<1%	<1%	1%	2%	1%
EXECUTE CHILD	9%	12%	16%	16%	17%	41%	40%	24%
CLEAN UP CHILD	<1%	<1%	<1%	<1%	<1%	< 1%	<1%	<1%
PROFILING	1%	1%	1%	1%	<1%	1%	1%	<1%

Table 2. *Speculative thread overhead breakdown.* Helper SpMT threads execute in a loop, idling for an opportunity to dequeue children from the priority queue, and then initialize them, execute them, and clean them up. The child execution process itself consists of startup, querying the return value predictor, useful work (currently only bytecode execution), and shutdown, induced by reaching some termination condition. There is profiling overhead both when executing speculative code, and when switching between tasks in the helper loop.

The striking result in Table 1 is that the parent spends so much of its time forking and joining speculative threads that its opportunities for making progress through normal Java bytecode and native code execution are reduced by up to 5-fold. We see that joining threads is significantly more expensive than forking threads, and that within the join process, predictor updates and waiting for the speculative child to halt execution are the most costly subcategories. Other overhead sub-categories are not insignificant, and

in general, optimisations to any of them will improve performance. The cost of profiling is high, but disappears in a final build.

In Table 2, we can make several observations about the execution of speculative children. First, the SpMT helper threads spend the majority of their time being idle, waiting to dequeue tasks from the priority queue, the implication being that the queue is often empty. In these experiments, we allow for out-of-order spawning [31], in which multiple *immediate* children are attached to a non-speculative parent, one per Java stack frame. However, we do *not* allow for speculative children to fork speculative children of their own, which greatly limits the available parallelism. Once supported, we will use the same profiling system to analyse the effect that forking several generations of children has on both parent and child overhead, and hope to see a decrease in the number of idle helper cycles.

We also note that when the helper threads *are* running speculative children, they spend a majority of their time doing useful work, which is bytecode execution only for speculative threads; in fact, if idle times are ignored, more cycles are spent doing useful work speculatively than non-speculatively. Outside of bytecode execution, we see that predictor lookup is quite expensive, most likely because of synchronization on dynamically expanding hashtables.

4.3.2 Parallelism Analysis

SableSpMT allows for investigation into runtime speculative parallelism at a fairly fine granularity. Speculative thread lengths are recorded on a per-callsite basis and can be analysed in both the single-threaded simulation (ST) and multithreaded (MT) modes. Thread length information, particularly when associated with specific callsites, can be quite instructive as to the effect of SpMT optimisations on the system. In the ST mode, children run until either an unsafe operation occurs or an arbitrary limit on sequence length is reached. Using a sequence length limit of 1000 instructions, we find that over all speculative children, 30% are successful and in the 0–10 bytecode instructions range, with very few failures, and 15% are successful and run for 90+ instructions. On the other hand, 25% of all threads are accounted for by failures at 90+, which derives from the correspondance between thread length and risk of dependence violation or unsafe execution [28].

In the MT mode, child threads are also stopped when parents return to fork points or pop frames in the exception handler, and 80% of speculative threads are accounted for by success in the 0–10 range, with only 1–2% found in subsequent 10 instruction buckets. As we reduce overhead costs, we expect children to run longer, and for parallelism to increase. An interesting point to note is that in hardware simulations, thread lengths of 40 *machine* instructions are considered impressive [17], and although uncommon, some children in our MT mode can run for hundreds of *bytecode* instructions.

In Figure 8, we examine *speculative coverage*, that is, the percentage of sequential program execution that occurs successfully in parallel. Adding processors to the system has an effect on all benchmarks, and with just 4 processors and no support for speculative children of speculative children, the amount of parallel execution is quite high, nearly 33% on average. Disabling the RVP framework brings the average speculative coverage on four processors from 33% to 19%, and thus we can confirm the result previously obtained by Hu *et al.*, namely that RVP plays an important role in SMLP.

4.3.3 Runtime Profiling

SableSpMT provides a facility for runtime profiling and feedback-directed optimisation. This is often crucial to performance in runtime systems; for example, JIT compilers typically depend on interpreter profiling to determine hot execution paths [1]. We make various measurements of the dynamic performance of our system

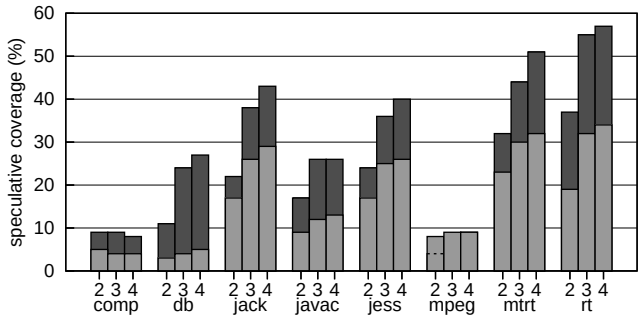

Figure 8. *Speculative coverage with and without RVP.* The SPECjvm98 benchmarks are shown running with 2, 3, and 4 processors, and the dark regions indicate the improvement as return value prediction is enabled.

available to optimisations by associating data with the fork and join points surrounding invokes. Currently our optimisations are written to exploit per-callsite information and thus context-sensitivity, although the data does generally remain available on a per-target basis.

In the context of return value prediction, our hybrid predictor selects the best sub-predictor over the last 32 return values, predictor hashtables expand according to load factors and performance, and future work will address disabling sub-optimal predictors after a warmup period. In the context of choosing optimal fork points, we assign child thread priorities or disable speculation completely according to various dynamic profiling data, including transitive method size, speculation success and failure, speculative sequence lengths, and premature child termination due to over-frequent forks.

It is worth noting that we have not discovered an optimal heuristic for forking new threads, although others have done work in this field on simulated hardware [41]. As we disable speculation at undesirable fork points, our system exhibits significant speedup, but the source of this speedup is both better opportunity for speculation as well as reduced overhead. Given that significant reductions in overhead are likely possible without reducing the number of dynamic forks, we defer this investigation until no further speedup can be made on that front.

4.3.4 Execution Times

experiment	`comp`	`db`	`jack`	`javac`	`jess`	`mpeg`	`mtrt`	`rt`	mean
SpMT must fail	1297s	931s	293s	641s	665s	669s	1017s	1530s	722s
SpMT may pass	1224s	733s	211s	468s	405s	662s	559s	736s	539s
relative speedup	1.06x	1.27x	1.39x	1.37x	1.64x	1.01x	1.82x	2.08x	1.34x
vanilla SableVM	368s	144s	43s	108s	77s	347s	55s	67s	120s
actual slowdown	3.33x	5.09x	4.91x	4.33x	5.26x	1.91x	10.16x	10.99x	4.49x

Table 3. *Execution times and relative speedup.* Geometric means do not include `raytrace` (`rt`), the single-threaded version of `mtrt`.

Although the focus of this work is in demonstrating the utility of SableSpMT as a research and analysis framework, measurable speedup remains the ultimate goal of any speculative system, and we provide overall performance data in Table 3. We compare our multithreaded results against executions where SpMT failure is artificially forced at every join point, and are thus able to provide a dynamic upper bound on speedup by factoring out overhead concerns. The geometric mean speedup over SPECjvm98 of normal speculation against a forced failure baseline is 1.34x. It is worth emphasizing again that as speculative threads become able to fork their own children, this upper bound is expected to expand. Although this approach is not perfectly accurate for obvious reasons,

our results do lean towards being somewhat pessimistic in terms of calculated speedup: Table 1 shows that SpMT failure is slightly less expensive than success, and the fact that we compute a speedup of only 1.01x for `mpegaudio` despite a speculative coverage of 9% derives from this.

Finally, we can compare the performance of our system against SableVM's vanilla switch interpreter upon which our framework is based. From an analysis perspective, experiments run in acceptable times, well suitable for normal, interactive usage. In general, these execution times are within one order of magnitude of sequential execution, and compare favourably with those of hardware simulations providing the same level of functionality and architectural detail, which can be within three orders of magnitude [20]. Clearly, however, there are significant improvements required in order to achieve actual speedup with our SMLP design. Our overhead analysis suggests a number of potential optimisations to reduce overhead and increase the relative amounts of speculative execution. Designing and implementing these further improvements is part of our future work.

5. Conclusions & Future Work

Investigation of any sophisticated, general optimisation strategy requires significant design, implementation and experimental flexibility, as well as a common ground for investigation. Our system provides a robust framework for Java SpMT exploration that simplifies the implementation effort and allows for easy data gathering and analysis. We include detailed collection of dynamic data, but also allow for application of internal feedback at runtime. To evaluate high level program information we include an interface to Soot-generated Java attributes, and can thus incorporate static information as well. We have demonstrated the use of all these features through realistic optimisation and performance analyses.

Measurements of speculative sequence length, speculative coverage, and relative speedup all indicate that significant parallelism does exist in the sequential threads of Java programs, and our analysis of speculation overhead indicates where to focus optimisation efforts. We are relatively optimistic as to improving the efficiency of our initial SpMT implementation, and actual speedup is a goal for the very near future.

Continued improvements to our framework will also provide for new research opportunities. We have implemented method level speculation, but other researchers have also had success with loop level [32], lock level [24, 30], and arbitrary speculation [2] strategies. These approaches have largely common internal requirements, and side-by-side implementations within our framework will make direct and meaningful comparisons of the various techniques feasible, and furthermore enable their composition.

Although the speculation support components in SableSpMT are fairly modular, they are integrated with SableVM and do not always provide an abstract interface. As part of our future work, we plan to move these components into a separate and language independent runtime support library for SpMT, `libspmt`. Features it will provide include control of SpMT helper threads, priority queueing, return value prediction, dependence buffering, stack buffering, fork heuristics, and various dynamic analysis facilities. Transforming multiple VMs and compilers to use this library will help to define an appropriate API.

Significant further work is required, but providing JIT compiler support for software SpMT is the next major challenge. The presence of a JIT offers both positive and negative opportunities for SpMT analysis and execution, and will certainly be interesting to examine. We are currently working on reusing elements of our design to support SpMT in IBM's Testarossa JIT and J9 VM.

Acknowledgements

We would like to thank Etienne Gagnon for his help in SableVM development, and in particular for suggesting the single-threaded simulation mode. We would also like to thank our colleagues Laurie Hendren, Allan Kielstra, and Mark Stoodley for constructive criticism of initial drafts of this paper. This research was funded by an IBM CAS fellowship, NSERC, FQRNT, and McGill University.

References

[1] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, Feb. 2005.

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

[3] 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.

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

[5] 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 '03)*, pages 434–446, June 2003.

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

[7] J. J. Cook. Reverse execution of Java bytecode. *The Computer Journal*, 45(6):608–619, May 2002.

[8] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*, pages 71–81, June 2004.

[9] B. Dufour. Objective quantification of program behaviour using dynamic metrics. Master's thesis, McGill University, Montréal, Québec, Canada, Oct. 2004.

[10] 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.

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

[12] R. Friedman and A. Kama. Transparent fault-tolerant Java virtual machine. In *Proceedings of the 22nd IEEE Symposium on Reliable Distributed Systems (SRDS '03)*, pages 319–328, Oct. 2003.

[13] E. M. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, McGill University, Montréal, Québec, Canada, Dec. 2002. `http://www.sablevm.org`.

[14] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, Mar. 2000.

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

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

[17] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*, pages 59–70, June 2004.

[18] 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 '00)*, pages 559–564, May 2000.

[19] I. H. Kazi and D. J. Lilja. Coarse-grained thread pipelining: A speculative parallel execution model for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 12(9):952–966, Sept. 2001.

[20] 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 '98)*, pages 286–293, Oct. 1998.

[21] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, Sept. 1999.

[22] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, 2nd edition, 1999.

[23] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*, pages 378–391, Jan. 2005.

[24] J. F. Martínez and J. Torrellas. Speculative locks for concurrent execution of critical sections in shared-memory multiprocessors. In *Proceedings of the 1st Annual Workshop on Memory Performance Issues (WMPI '01)*, June 2001.

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

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

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

[28] C. J. F. Pickett and C. Verbrugge. Speculative multithreading in a Java virtual machine. Technical Report SABLE-TR-2005-1, Sable Research Group, McGill University, Mar. 2005.

[29] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. In R. Wilhelm, editor, *Proceedings of the 10th International Conference on Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science (LNCS)*, pages 334–354, Apr. 2001.

[30] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 34)*, pages 294–305, Dec. 2001.

[31] 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 *Proceedings of the 19th ACM International Conference on Supercomputing (ICS '05)*, June 2005.

[32] P. Rundberg and P. Stenström. An all-software thread-level data dependence speculation system for multiprocessors. *Journal of Instruction-Level Parallelism*, 3, Oct. 2001.

[33] The SPEC JVM Client98 benchmark suite. `http://www.spec.org/jvm98/jvm98/`.

[34] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*, pages 1–12, June 2000.

[35] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 2005. To appear.

[36] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliaro, and S. S. Tse. The MAJC architecture: A synthesis of parallelism and scalability. *IEEE Micro*, 20(6):12–25, Dec. 2000.

[37] J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, Sept. 1999.

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

[39] F. Warg and P. Stenström. Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT '01)*, pages 221–230, Sept. 2001.

[40] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '05)*, Oct. 2005. To appear.

[41] J. Whaley and C. Kozyrakis. Heuristics for profile-driven method-level speculative parallelization. In *Proceedings of the 34th International Conference on Parallel Processing (ICPP '05)*, pages 147–156, June 2005.

[42] K. Yoshizoe, T. Matsumoto, and K. Hiraki. Speculative parallel execution on JVM. In *Proceedings of the 1st UK Workshop on Java for High Performance Network Computing*, Sept. 1998.