Supporting Higher-Order Virtualization

Jeremy Singer, Chris Kirkham, and Ian Watson

School of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, UK.
{jsinger,chris,watson}@cs.man.ac.uk

Abstract. Virtualization is ubiquitous, with the global availability of the Java Virtual Machine and other similar virtual machine platforms. Higher-order virtualization involves building a stack of virtual machine layers. This provides obvious advantages such as: flexibility; separation of concerns; reuse of existing functionality; support for legacy platforms. However, the benefits of higher-order virtualization come at a price in terms of efficiency. This paper examines different techniques to increase the efficiency of higher-order virtualization on chip multiprocessor architectures. These techniques embrace hardware, software and virtual machine interaction. Some techniques (such as just-in-time compilation) are employed in existing virtual machine systems. Other techniques (such as hints-based speculative parallelism) have not been previously considered. We examine how to use these performance-enhancing techniques in the context of stacked virtual machine layers.

1 Introduction

This paper introduces the subject of *higher-order virtualization* (HOV), which involves creating a stack of virtual machine (VM) layers for program execution. Each self-contained layer may address a different aspect of the system, in the same way that each layer of a network protocol stack addresses a different communication issue. The individual layers are generally small enough to facilitate software maintenance and re-engineering. The HOV phenomenon is not new. For instance, Smith and Nair [SN05] and Be Dope [Dop98] describe examples of VMs executing on top of other VM environments. However until now, HOV has not been considered as a discipline in its own right. This paper presents some guiding principles. It explores the tension between efficiency and isolation of adjacent layers in the stack. It highlights the recent growth in popularity of both virtualization (with systems such as Java and .NET) and parallelism (with architectural features such as hyperthreading and dual-core). There is a (perhaps accidental) synergy between virtualization and parallelism. This synergy should be exploited in the context of HOV, in order to amortize inefficiency.

The HOV principles presented in this paper are demonstrated by means of a simple, proof-of-concept, stacked VM system. Figure 1 illustrates the system. A primitive stack-based bytecode interpreter executes on top of an adaptive parallelizing Java VM, which executes on top of an experimental (simulated) chip multiprocessor (CMP) platform. (Note that, in a sense, the simulator is also a VM that executes on top of real hardware, namely a Linux IA32 system. However for the purposes of this paper, we ignore the fact that the CMP is simulated, and treat it as though it were the physical layer. This is a compelling beauty of the HOV view. When one takes a top-down view of the system, it is possible to be myopic!)

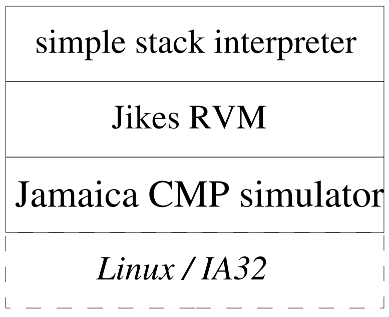

Fig. 1. Overview of HOV system (stacked VM)

Execution speed of bytecode interpreted by the top-level VM can be improved using standard interpretive techniques as well as standard parallel techniques. At each stage, the top-level interpreter uses hooks provided by the host JVM to communicate information directly to the lower-level. In our case, the host JVM is Jikes RVM, which provides a disciplined API to break through the abstracted Java view of the world. This approach is similar to the *paravirtualization* technique for virtualized OSs [BDF$^+$03]. OSs that are to be paravirtualized require source code changes to hook into the underlying host VM monitor. In the same way, our top-level interpreter requires source code changes to hook into the underlying host JVM.

1.1 Contributions

This paper makes four contributions.

1. It introduces, motivates and clarifies the concept of higher-order virtualization (HOV).
2. It highlights the synergy between virtualization and parallelism. This synergy should be exploited for HOV.
3. It provides techniques for improving the efficiency of higher-order virtualized systems, particularly in the context of a demonstration interpreter running atop Jikes RVM on the experimental Jamaica CMP platform.
4. It discusses the uneasy tension between efficiency and isolation of stacked VM layers. It advocates a middle way, that exposes some host features to

the guest. However that exposure is carefully encapsulated by a well-defined API, which acts as a managed trapdoor to lower-level features.

1.2 Outline

The rest of this paper is structured as follows. Section 2 details the motivation for HOV, from both software and hardware considerations. Section 3 describes implementation techniques to improve the efficiency of a stack-based interpreter, particularly in the context of an HOV system. Section 4 outlines the prototype system used to demonstrate the principles outlined in this paper. Section 5 presents results from the prototype system. These results should extrapolate to larger, more realistic HOV systems. Section 6 examines related work. Finally Section 7 concludes.

2 Motivation

Higher-order virtualization (HOV) abstracts a virtual entity, rather than a physical entity. In the context of HOV for runtime environments, there are numerous motivating factors for creating a stack of VM layers.

2.1 Software engineering reasons

Separation of concerns: In the same way that different layers of a network protocol stack provide different facilities (such as error correction) different layers of a virtual machine stack can provide different facilities. This paper introduces a VM layer that builds speculative parallelism on top of traditional thread-level parallelism. In this way, speculation can be implemented easily (virtually!) on a platform that does not support speculative execution natively.

Portability: This is the greatest advantage of virtualization. When retargetting to a new platform, only the platform-specific components need to be ported, since the remainder of the virtualized code does not contain platform-specific assumptions. Of course there is an additional consideration in the case of a stacked VM system, since VM layers may also migrate up or down the stack. Portability is no longer merely a *horizontal* issue (change of underlying platform) but also *vertical* (move VM up or down the stacked VM system).

Legacy code support: Legacy platforms can be virtualized and run as part of a stacked VM system. Such VMs can use facilities provided by lower layers (such as sandboxing to localize program failures).

Flexibility: In an HOV system, user applications should be able to run at any level of the system. Some applications may run at high levels, taking advantage of abstractions provided by underlying VM layers. Other applications may run at lower levels, for reasons of efficiency or other application-specific requirements that cannot be satisfied by higher-level layers. Flexibility provides choice and tradeoffs for user applications.

2.2 Architectural reasons

Until now, HOV has not been feasible due to its inefficiency. As with all abstraction, extra layers incur execution overhead. This paper shows that some of the overhead may be eliminated. However, the tremendous processing power of modern commodity hardware ensures that there is scope for a measure of residual inefficiency.

Another recent advance is the widespread availability of parallel resources. Parallelism (that may otherwise be redundant) is able to soak up some of the inefficiency introduced by HOV, as this paper demonstrates.

The underlying message is that it is feasible to pay a price in terms of performance in order to improve usability. Since physical processor cycles are so cheap, we can afford to expend many of them in the quest for virtualized processor cycles.

3 Interpretation Techniques

The two opposite ends of the spectrum of bytecode execution environments are *interpretation* and *just-in-time compilation* (JIT). In general, interpretation executes bytecode programs from 10 to 1000 times slower than native code [EGKP02]. On the other hand, JIT executes bytecode programs at near-equivalent speeds to compiled code executing directly on the host platform [Ayc03]. Clearly execution speed is the major advantage of JIT. One advantage of interpretation is that the interpreter needs to know almost nothing about its host platform. This clear abstraction layer between host and guest facilitates interpreter portability. In contrast, the JIT system needs to know almost everything about its host platform, in order to generate correct code. The abstraction layer between host and guest is rather blurred, if it exists at all.

This study focuses entirely on interpreters, since they are suitable for rapid prototyping and easy experimentation. Section 3.1 describes a simple interpreter model that is completely insulated from the host platform. Then Sections 3.2 and 3.3 show how bytecode execution speed can be increased by the exposure of specific host services to the interpreter. Finally Section 3.4 introduces speculative parallelism, which requires further communication between host and guest VMs. At this stage, the interpreter model is gradually shifting towards the JIT end of the spectrum of bytecode execution environments outlined above.

Note that since the top-level interpreter runs on a host JVM, the implementation source language is required to be Java. Thus all the techniques for improving interpreter performance must be expressible in Java syntax.

3.1 Switch-based interpretation

The simplest interpreter has an outer while loop that steps through the bytecodes at the interpretive program counter. In each iteration of the loop the interpreter executes a giant switch statement, with a different case for each bytecode operation. Operands reside on a memory stack. Operations take place directly on the top of the stack. Figure 2 shows some example source code for a switch-based interpreter.

```
while (pc < MAXPC) {
    opcode = getByteAt(pc++);
    switch(opcode) {
    case Bytecodes.ADD:
        a = pop();
        b = pop();
        push(a+b);
        break;
    // ...
}
```

Fig. 2. Source code for switch-based interpreter

Advantages: Very easy to implement, modify and debug. Portability.
Disadvantages: Very slow execution, due to *control-flow* inefficiencies (direct and indirect branch per bytecode dispatch) and *data-flow* inefficiencies (stack-based computation induces many memory accesses).

3.2 Threaded interpretation

The switch-based interpretation model involves convoluted control flow. Each time a bytecode instruction is dispatched, execution flows through the single fetch-decode sequence at the start of the while loop body, then it branches to the correct bytecode case in the switch statement. This incurs one direct branch and one indirect branch. Moreover, the single indirect branch instruction is used for branching to all case statements, so branch prediction hardware is unlikely to perform well here A better approach is to duplicate the fetch-decode sequence at the end of each case, enabling dispatch to take place with a single indirect branch. This is known as *threaded interpretation* [Bel73]. It removes the direct branch to the start of the while loop body. Also there are now lots of indirect branch instructions (one for each case) and since some instruction sequences are more likely than others, branch prediction hardware may perform well. Ertl and Gregg provide evidence to support this claim [EG03]. Unfortunately, the Java language does not support this style of indirect branching. We overcome the problem by exposing some host VM services that enable indirect branching. These services were originally intended for dynamic linking after JIT compilation of methods. We make use of the Jikes RVM `VM_Magic` API to make dynamic jumps to methods, where each method now corresponds to a switch case from the original interpreter. Figure 3 shows some example source code for a threaded interpreter.

```
public static void start() {
    int opcode = getByteAt(pc++);
    VM_CodeArray nextCall = offsets[nextOpcode];
    VM_Magic.dynamicBridgeTo(nextCall);
}

public static void bc_ADD() {
    if (numCached == 0) {
        x = mem[sp/4];;
        sp-=4;
        y = mem[sp/4];
        sp-=4;
        first = x + y;
        numCached = 1;
    }
    else if (numCached == 1) {
        y = mem[sp/4];;
        sp-=4;
        first = first + y;
        // numCached = 1;
    }
    else if (numCached == 2) {
        first = first + second;
        numCached = 1;
    }
    else {
        // ...
    }
    nextOpcode = getByteAt(pc++);;
    VM_CodeArray nextCall = offsets[nextOpcode];
    VM_Magic.dynamicBridgeTo(nextCall);
}
```

Fig. 3. Source code for threaded interpreter with stack-caching and method inlining

Advantages: Increase in execution speed.
Disadvantages: Reduced portability. Requires interfacing to host API for indirect branches. Increased code duplication (impact on code size and maintainability).

3.3 Stack-caching and method inlining

There are several straightforward transformations that can be performed on the threaded interpreter to improve its efficiency, without exposing further host features. The top few stack elements may be *cached* in register-allocated local variables [Ert95]. This reduces the number of memory accesses (which are slow). Since most operations take place on top of stack, these will now be register-register operations rather than memory-memory. Also, it is possible to *inline* method calls such as stack push and pop operations. Method calls can be expensive. It is often cheaper to execute inlined instruction sequences. Figure 3 shows some example source code for a threaded interpreter with stack-caching and method inlining.

Advantages: Simple improvements lead to increased efficiency.
Disadvantages: Increased code duplication causes code size increase and maintainability problems (unless non-standard Java macro-expansions are used).

3.4 Speculative parallel interpretation

Bytecode interpretation is an inherently sequential computation. The fetch-decode-execute scenario does not readily admit parallelism, especially with stack-based computation. However, there may be latent parallelism in the bytecode program. If the host VM is able to support parallel computation, then the interpreter needs to somehow communicate this information to the host. Extra *annotation* bytecodes convey hints to the host about regions of bytecode that can be interpretively executed in parallel. Such an annotation bytecode may occur immediately before a loop that may be executed in parallel. When the interpreter encounters this annotation bytecode during execution, it sends a message to the host VM to indicate that more threads should be forked to interpret the loop iterations in parallel. Each forked thread runs until it has finished executing its portion of the loop iterations. Then a single thread continues sequential interpretation.

One novelty of our approach is that we attempt to build support for *thread-level speculation* directly in the interpretation layer. The idea is that the interpreter forks several interpretation threads at speculative parallel execution points. The underlying host VM does not need to know which threads are speculative and which threads are not. The interpretation system manages the speculation itself, by either committing or rolling back the effects of speculative interpreter threads. The interpretation system can use standard memory access buffering techniques to enforce correct speculative execution with support for roll-back in the case of data dependence violations.

There is scope for two-way communication, since the host VM can send a reply back to the interpreter when insufficient parallel resources are available. This paradigm is *hints-based speculative parallelism* which should become increasingly useful to support effective HOV. Figure 4 shows some example source code for an interpreter that supports speculative parallelism over loops.

Advantages: Increase in execution speed. Effective deployment of parallel resources that are not usable by ordinary sequential interpretation techniques.
Disadvantages: Further exposure of host-specific services, which reduces portability. (However note that the interpreter only gives *hints* to the host VM, so it is possible to ignore these hints.)

3.5 Improving interpreter performance

The performance of the interpreter improves as the interpreter model changes from switch-based (Section 3.1) to threaded (Section 3.2) to parallel (Section 3.4). This performance increase is clear to see from the fetch-decode-execute (FDE) code layout, as shown in Figure 5. The switch-based model has a single FDE code sequence, and all bytecode interpretation goes through this sequence. The threaded model has multiple FDE code sequences, one for each instruction. A single instruction is in-flight at once. The parallel model has multiple FDE code sequences in-flight at once.

4 Experimental Setup

This section explains the details of the VM stack used for the experiments.

4.1 Jamaica architecture

The Jamaica architecture is the basis of our research into compiler support for chip multiprocessors (CMPs). The experimental processor structure is a CMP constructed of simple 5-stage pipeline cores. An important aspect of the hardware is the ability to create and distribute lightweight threads with minimum overheads. This is supported by a *free thread* mechanism, in which the availability of free hardware threads is signalled by the passing of a token on a separate ring bus [WEMW02]. This is coupled with a register windows based ISA which facilitates the maintenance of thread context when performing thread switches. This is particularly useful when implementing multiple thread contexts per core to permit thread switching on cache misses to hide latency. Because of the individual nature of the ISA, the cores have not been based on any specific processor. The instruction set is similar to that of an Alpha but with a number of extensions and modifications. The experimental simulator is written in C. The number of processor cores and hardware contexts-per-core can be specified at simulator startup. All experiments reported in this paper use a 4-core, 2-contexts-per-core layout. Recent experiments have verified that a single core can be simulated at an approximate instruction execution rate of 1.5 MIPS on a 3GHz IA32 Linux workstation.

```
public static void setupSpecLoop() {
  // get parameters from subsequent bytecodes
  // start value for loop iterator
  int begin = pop();
  // end value for loop iterator (not inclusive)
  int limit = getWordAt(ThreadedBCs.pc);
  ThreadedBCs.pc += 4;
  // increment value for loop iterator (signed)
  int step = getWordAt(ThreadedBCs.pc);
  ThreadedBCs.pc += 4;
  // pc-relative offset to code after loop
  int offset = getWordAt(ThreadedBCs.pc);
  ThreadedBCs.pc += 4; // now pc points to 1st instr in loop

  // now do speculation, simple example has 2
  // hard-wired speculative interpreter threads

  // first thread
  ThreadedBCs_1.pc = ThreadedBCs.pc;
  ThreadedBCs_1.fp = ThreadedBCs.fp + NEW_THREAD_STACK;
  // ...
  VM_Magic.branchCall();  // thread migrates to diff processor
  startSpec1();

  // and in parallel, second thread
  ThreadedBCs_2.pc = ThreadedBCs.pc;
  ThreadedBCs_2.fp = ThreadedBCs.fp + 2*(NEW_THREAD_STACK);
  // ..
  VM_Magic.branchCall();  // thread migrates to diff processor
  startSpec2();

  // now both speculative interpretation threads
  // should be running in parallel, wait
  // for them to complete
  while (VM_Magic.branchThreadSynchronize() != 0) {
    Thread.yield();
  }

  // ...
}
```

Fig. 4. Source code for speculative parallel interpreter

model	*static FDEs*	*dynamic FDEs*
switch-based	one	one
threaded	many	one
parallel	many	many

Fig. 5. Fetch-decode-execute (FDE) characteristics of different interpreter models

4.2 Jikes RVM

The IBM Jikes RVM [AAB$^+$00,AAB$^+$05] is written mostly in Java and there is a great deal of published material about its structure. In addition, there is a very active research community experimenting with many aspects of dynamic compilation and more general VM issues. A version of the Jikes RVM has therefore been constructed on top of the instruction-level simulator. This required both the retargetting of the compilers and the extension of the system to make use of multiple cores. This includes the mapping of Java threads onto multiple cores and the handling of concurrent garbage collection.

Although the Jikes RVM has been designed for the execution of Java programs there are many aspects of the system that are applicable to the execution more general programs. Java bytecode is translated to a high-level intermediate representation (HIR). This is a 3-address format which enables a range of optimisations to be performed. As the dynamic compilation proceeds, this is translated through two further intermediate forms before reaching the native code of the target processor. At the highest level, general optimisations such as loop unrolling are performed. This is the level at which we are implementing generic (non-speculative) parallelization optimisations. At lower levels, more processor specific optimisations such as register allocation are done.

The Jikes RVM executes about 500 million instructions in order to boot itself ready to execute a basic Java program; that is about 10 minutes of simulated execution. A program such as the `_205_raytrace` SPECjvm98 benchmark, which executes for approximately 15 seconds on a 3GHz IA32 requires about 10 hours of simulation time for a single Jamaica processor.

4.3 Interpreter

We designed a simple stack-based bytecode virtual instruction set, influenced by PCode [Nel79] and OCode [Ric71]. It has standard stack-based operations for integer arithmetic, conditional branches, memory access and function calls. It also has some support for speculation, as described later in this section. Figure 6 reviews the full instruction set.

We developed a series of simple interpreter VMs for this stack-based virtual instruction set. All the interpreter models are developed in the Java programming language. The simplest model is a switch-based bytecode interpreter, conforming to the description in Section 3.1. This simplest model can run on top of

operation classification	*instruction opcodes*
integer arithmetic operations	`ADD`, `SUB`, `MUL`, `DIV`, `MOD`
conditional branch operations	`B`, `BEQ`, `BNE`, `BLT`, `BLE`, `BGT`, `BGE`
memory access operations	`LDW`, `STW`
function call operations	`CALL`, `RET`
stack manipulation operations	`DUP`, `DUPx2`, `SWAP`, `POP`, `PUSHFP`, `PUSHCONST`
speculation support annotations	`SPECLOOP`, `SPECLBEGIN`, `SPECLEND`

Fig. 6. Bytecode operations for Simple Interpreter

any standard JVM. More complicated interpreter models (as described in Sections 3.3 to 3.4) make assumptions about the services provided by the underlying host VM. These services are specifically provided by the `VM_Magic` API of the Jamaica port of the Jikes RVM.

The most complicated interpreter model supports speculative parallel interpretation. In this model, annotation bytecodes mark the start and end of each loop body that may be speculatively parallelized. These annotations must be inserted automatically by a static compiler, or manually at the assembly language level. The annotation information gives static bounds on loop iterator values. Note that this information is only a suggestion about speculation. It can be disregarded, indeed, it is ignored entirely by non-parallel interpreter models. Unfortunately, due to time constraints, at present our speculative parallel interpreter model is not fully functional. It buffers memory accesses by speculative threads, but it is unable to support rollback in the case of data dependence violations.

4.4 Test programs

There are two microbenchmark programs to test the interpreter. The `primes` program checks the primality of integers and does some simple mathematical computations based on the outcome. The `matrix` program performs matrix multiplication for matrices with integer elements. Since this system is still at the prototype stage, there is no compiler for the bytecode interpretive language. Both the microbenchmarks were hand-coded.

5 Results

All the reported results are in terms of cycle counts. These are obtained from the instruction-level simulator of the Jamaica architecture. Figure 7 shows the cycle counts taken by different interpreter models on the two microbenchmarks. Due to time constraints we were unable to obtain a cycle count for the `primes` program on the speculative parallel interpreter.

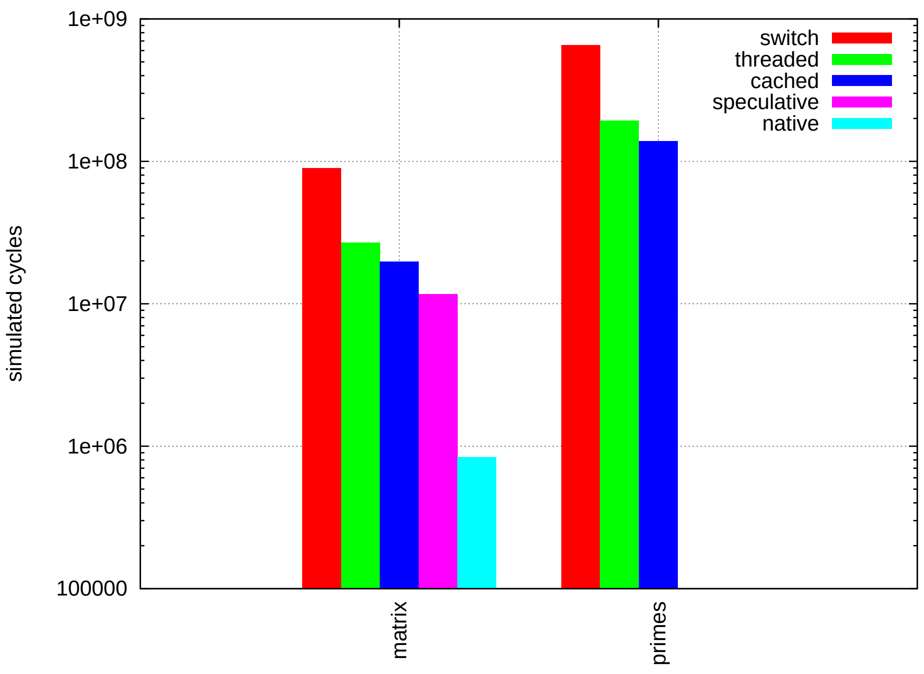

Fig. 7. Simulated cycle counts for different interpreter models running the microbenchmark programs

We also coded the `matrix` program in Java and ran it "natively" on the Jamaica Jikes RVM, to compare cycle counts. The native version took just over 840,000 cycles. If we normalize the results for `matrix` in relation to this native result, then the execution times are as follows.

execution system	*relative time*
native	1
speculative parallel	14
cached/inlined	24
threaded	32
switch-based	107

These results show that increasingly sophisticated interpreter models get closer to the performance of native code, but are still over 10 times slower than it.

For the record, the Jamaica Jikes RVM configuration used is `OptBaseNoGC`. All cycle counts were obtained on the second run of the interpretation, presuming all Jikes RVM compilation overhead to be factored into the first run of the interpretation.

6 Related Work

6.1 Interpretation techniques

Klint [Kli81] gives the earliest survey paper for interpretation techniques. Ertl and Gregg [EG03] provide motivation for efficient interpretation, and then describe some techniques. However neither of these surveys includes speculative techniques. Pickett and Verbrugge [PV05] describe the implementation of a speculative interpretation system for Java, running on top of a native processor rather than another VM. They employ method-level speculation, which predicts return values from method calls before they have completed. Yoshizoe et al [YMH98] use loop-level speculation, which predicts loop variant values before loop iterations have completed. Again, this is for a Java interpreter running on a native platform. Our approach is different because our interpreter VM only provides hints to the underlying Jamaica Jikes RVM. These other interpreters make explicit decisions, since they run directly on non-virtual platforms.

6.2 VM abstraction techniques

Piumarta et al [PFSB00] advocate the *virtual virtual machines* approach, which brings the concepts of aspect orientation to VMs. There are a wide range of different VM services, and the system dynamically weaves together as many services as necessary to create a VM with sufficient capabilities to run each application. This is a virtualization that broadens the functionality of a single VM, rather than building guest VM layers on top of host layers in a stack. The Xen approach of *paravirtualization* [BDF$^+$03] aims to multiplex around a hundred x86 VMs on a single system, with low virtualization overhead and pervasive resource accountability. This provides a farm of VMs, not a stack. The whole aim of Xen is to avoid interaction between these VMs altogether. The *parallel virtual machine* (PVM) [GBD+94] abstracts a heterogeneous cluster of machines, then permits client program to use these resources in parallel. In this sense PVM is similar to our system, however PVM allows client programs to specify precisely their parallelization policies, whereas our system only permits clients to make hints.

7 Concluding Remarks

The primary motivation for this research is that there is no speculation support in the Jamaica version of Jikes RVM at present. We have implemented a simple software speculation VM layer, which seems the easiest and least intrusive approach [CL03]. However, this approach has several disadvantages.

1. Software support is always less efficient than hardware support.
2. Existing Java programs cannot be candidates for speculation. Only bytecode programs executed by the simple interpreter can be speculatively parallelized.

Nevertheless this study was justified, for the reasons below.

1. It has presented the HOV paradigm.
2. It has argued that it is straightforward to build speculation support on top of non-speculative parallelism.
3. These ideas have been implemented in a simple prototype bytecode interpreter system.

Much work remains to be done. The speculative parallel interpreter model must be developed to enable the speculative rollback mechanism. Microbenchmarks will be required to measure the overhead of rollback. The primitive loop-level speculation could be improved. A compiler for this simple bytecode language would facilitate easier production of microbenchmark tests. The long term goal of our research is to implement support for speculation directly within the Jamaica version of the Jikes RVM. However this small case study should be a useful experience.

References

[AAB+00] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, Feb 2000.

[AAB+05] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes research virtual machine project: Building an open source research community. *IBM Systems Journal*, 44(2):1–19, Feb 2005.

[Ayc03] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[CL03] Marcelo Cintra and Diego R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–24, 2003.

[Dop98] Be Dope. The russian doll contest, 1998. `http://www.bedope.com/contests/contest1.html`.

[EG03] M. Anton Ertl and David Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, 2003.

[EGKP02] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen—a generator of efficient virtual machine interpreters. *Software: Practice and Experience*, 32(3):265–294, Mar 2002.

[Ert95] M. Anton Ertl. Stack caching for interpreters. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 315–327, 1995.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing.* MIT Press, 1994.

[Kli81] Paul Klint. Interpretation techniques. *Software: Practice and Experience*, 11(9):963–973, Sep 1981.

[Nel79] Philip A. Nelson. A comparison of PASCAL intermediate languages. *ACM SIGPLAN Notices*, 14(8):208–213, 1979.

[PFSB00] Ian Piumarta, Bertil Folliot, Lionel Seinturier, and Carine Baillarguet. Highly configurable operating systems: the VVM approach. In *ECOOP'2000 Workshop on Object Orientation and Operating Systems*, 2000.

[PV05] Christopher J.F. Pickett and Clark Verbrugge. Speculative multithreading in a Java virtual machine. Technical Report 2005-1, Sable Research Group, McGill University, Mar 2005.

[Ric71] Martin Richards. The portability of the BCPL compiler. *Software: Practice and Experience*, 1(2):135–146, 1971.

[SN05] J.E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes.* Morgan Kaufmann, 2005.

[WEMW02] Greg Wright, Ahmed El-Mahdy, and Ian Watson. *Java Microarchitectures*, chapter 10: Java Machine and Integrated Circuit Architecture (JAMAICA), pages 187–206. Kluwer, 2002.

[YMH98] K. Yoshizoe, T. Matsumoto, and K. Hiraki. Speculative parallel execution on JVM. In *First UK Workshop on Java for High Performance Network Computing*, 1998.