Context Threading: A flexible and efficient dispatch technique for virtual machine interpreters *

Marc Berndl, Benjamin Vitale, Mathew Zaleski and Angela Demke Brown
University of Toronto, Systems Research Lab
{*berndl,matz,bv,demke*}*@cs.toronto.edu*

Abstract

Direct-threaded interpreters use indirect branches to dispatch bytecodes, but deeply-pipelined architectures rely on branch prediction for performance. Due to the poor correlation between the virtual program's control flow and the hardware program counter, which we call the context problem*, direct threading's indirect branches are poorly predicted by the hardware, limiting performance. Our dispatch technique,* context threading*, improves branch prediction and performance by aligning hardware and virtual machine state. Linear virtual instructions are dispatched with* native calls and returns*, aligning the hardware and virtual PC. Thus, sequential control flow is predicted by the hardware return stack. We convert virtual branching instructions to native branches, mobilizing the hardware's branch prediction resources. We evaluate the impact of context threading on both branch prediction and performance using interpreters for Java and OCaml on the Pentium and PowerPC architectures. On the Pentium IV, our technique reduces mean mispredicted branches by 95%. On the PowerPC, it reduces mean branch stall cycles by 75% for OCaml and 82% for Java. Due to reduced branch hazards, context threading reduces mean execution time by 25% for Java and by 19% and 37% for OCaml on the P4 and PPC970, respectively. We also combine context threading with a conservative inlining technique and find its performance comparable to that of selective inlining.*

1 Introduction

Interpretation is a powerful tool for implementing programming language systems. It facilitates interactive program development and debugging, compact and portable deployment, and simple language prototyping. This combination of features makes interpreted languages attractive in many settings, but their applicability is constrained by poor performance compared to native code. Consequently, many important systems, such as Sun's HotSpot [1] and IBM's production Java virtual machine [21] run in mixed mode, compiling and executing some parts of a program while interpreting others. Baseline interpreter performance thus continues to be relevant.

Recently, Ertl and Gregg observed that the performance of otherwise efficient *direct-threaded* interpretation is limited by pipeline stalls and flushes due to extremely poor indirect branch prediction [5]. Modern pipelined architectures, such as the Pentium IV (P4) and the PowerPC (PPC), must keep their pipelines full to perform well. Hardware branch predictors use the *native* PC to exploit the highly-biased branches found in typical (native code) CPU workloads [10, 13]. Direct-threaded virtual machine (VM) interpreters, however, are not typical workloads. Their branches' targets are unbiased and therefore unpredictable [5]. For an interpreted program, it is the *virtual* program counter (or `vPC`) that is correlated with control flow. We therefore propose to organize the interpreter so that the native PC correlates with the `vPC`, exposing virtual control flow to the hardware.

We introduce a technique based on *subroutine threading*, once popular in early interpreters for languages like Forth. To leverage return address stack prediction. we implement each virtual instruction as a subroutine which ends in a native *return* instruction. Note, however, that these subroutines are not full-fledged functions in the sense of a higher-level programming language such as C (no register save/restore, stack frame creation, etc.). When the instructions of a virtual program are loaded by the interpreter, we translate them to a sequence of call instructions, one per virtual instruction, whose targets are these subroutines. Virtual instructions are then dispatched simply by natively executing this sequence of calls. The key to the effectiveness of this simple approach is that at dispatch time, the native PC is perfectly correlated with the virtual PC. Thus, for non-branching bytecodes, the return address stack in modern processors reliably predicts the address of the next bytecode to execute. Because the next dynamic instruction is

*This research was supported by NSERC, IBM CAS and CITO.

not generally the next static instruction in the virtual program, branches pose a greater challenge, For these virtual instructions, we provide a limited form of specialized inlining, replacing indirect with relative branches, thus exposing virtual branches to the hardware's branch predictors.

We review techniques for virtual instruction dispatch in interpreters, describe their performance problems, and define the *context problem* in Section 2. Then, we discuss other work on improving dispatch performance in Section 3. We provide relevant details of our implementations in a Java virtual machine (SableVM) and the OCaml interpreter on the Pentium IV and Power PC architectures in Section 4. Using an extensive suite of benchmarks for both Java and OCaml, we evaluate context threading in Section 5.

This paper makes the following contributions:

- We introduce a new dispatch technique for virtual machine interpreters that dramatically improves branch prediction and demonstrate that our technique does not depend on a specific language or CPU architecture.
- We show context threading is effective. On both the P4 and the PPC970, it eliminates 25% of the mean elapsed time of Java benchmarks, with individual benchmarks running twice as fast as with direct threading. For OCaml, we achieve a 20% reduction in the mean execution time on the P4 and a 35% reduction on the PPC970 with some benchmarks achieving as much as 40% and 50%, respectively.
- We show that context threading is compatible with inlining, using a simple heuristic that we call *Tiny Inlining*. On OCaml, we achieve speedups relative to direct threading of at least 10% over context threading alone. On Java, we perform as well as or better than SableVM's implementation of selective inlining.

2 The Context Problem

An interpreter executes a virtual program by dispatching its virtual instructions in sequence. The current instruction is indicated by a virtual program counter, or `vPC`. The virtual program consists of a list of virtual instructions, each consisting of an opcode and zero or more operands. The exact representation of the virtual program depends on the dispatch technique used.

A typical switch dispatched interpreter, implemented in C, is a `for` loop which fetches the opcode at `vPC`, and then executes a `switch` statement, each opcode being implemented by a separate `case` block, (the *opcode body*, or *body*). However, `switch` dispatch is considerably slower than the start-of-the-art, direct-threaded dispatch [7].

As shown in Figure 1, a direct threaded interpreter represents virtual program instructions as a *list of addresses*.

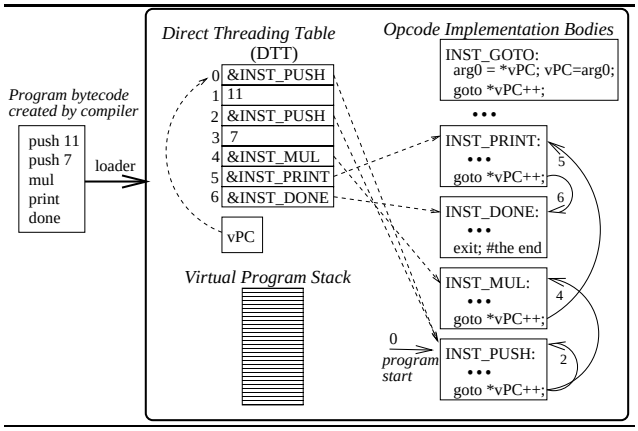


Figure 1. Direct Threaded VM Interpreter

```
mov eax = (rvPC)          lwz r0 = 0(rvPC)
addl 4,rvPC               mtctr r0
jmp *eax                  addi rvPC,rvPC,4
                          bctr
```

(a) Pentium IV assembly | (b) Power PC assembly

Figure 2. Direct Threaded Dispatch

Each address points to the opcode body. We refer to this list as the Direct Threading Table, or DTT. The operands are also stored in this list, immediately after the corresponding opcode address. The `vPC` points into the DTT to indicate the instruction to be executed. Note that, for each body, there are potentially many virtual instructions using that body. In the figure, for example, both INST_PUSH instructions point to a single body. The actual dispatch to the next instruction is accomplished by the `goto *vPC++` at the end of each opcode body, which is supported by GNU C's *labels-as-values* extensions. In Figure 2, we show the assembly code corresponding to this dispatch statement for the Pentium IV and PowerPC architectures.

When executing the indirect branch in Figure 2(a) the Pentium IV will speculatively dispatch instructions using a predicted target address. The PowerPC uses a different strategy for indirect branches, as shown in Figure 2(b). First the target address is loaded into a register, and then a branch is executed to this register address. Rather than speculate, the PowerPC stalls until the target address is known, although other instructions may be scheduled between the load and the branch to reduce or eliminate these stalls.

Stalling and incorrect speculation are serious pipeline hazards. To perform at full speed, modern CPU's need to keep their pipelines full by correctly predicting branch targets. Indirect branch predictors assume that the destination of an indirect branch is highly correlated with the address of the branch instruction itself. As observed by Ertl [5, 6], this assumption is usually wrong for direct threaded interpreter workloads. In a direct-threaded implementation, there is only *one* jump instruction per virtual opcode imple-

mented. For example, in Figure 1, there are two instances of INST_PUSH. In the context of vPC=0, the dispatch at the end of the INST_PUSH body results in a native indirect branch back to the start of the INST_PUSH body (since the next virtual instruction at vPC=2 is also an INST_PUSH). However, the target of the same native indirect branch in the context of vPC=2 is determined by the address stored at vPC=4, which in this example is an INST_MUL opcode. Thus, the target of the indirect branch depends on the *virtual context*—the vPC—rather than the *hardware pc* of the branch, causing the hardware to speculate incorrectly or not at all. We refer to this lack of correlation between the native PC and the vPC as the *context problem*.

3 Related Work

Much of the work on interpreters has focused on the dispatch problem. Kogge [12] remains a definitive description of many threaded code dispatch techniques. These can be divided into two broad classes: those which refine the dispatch itself, and those which alter the bodies so that there are more efficient or simply fewer dispatches. Switch and direct threading belong to the first class, as does subroutine threading, discussed next. Later, we will discuss superinstructions and replication, which are in the second class. We are particularly interested in subroutine threading and replication because they both provide context to the branch prediction hardware.

Some Forth interpreters use subroutine-threaded dispatch. Here, the program is not represented as a list of body addresses, but instead as a sequence of native *call*s to the bodies, which are then constructed to end with native *return*s. Curley [3, 4] describes a subroutine-threaded Forth for the 68000 CPU. He improves the resulting code by inlining small opcode bodies, and converts virtual branch opcodes to single native branch instructions. He credits Charles Moore, the inventor of Forth, with discovering these ideas much earlier. Outside of Forth, there is little thorough literature on subroutine threading. In particular, few authors address the problem of where to store virtual instruction operands. In Section 4, we document how operands are handled in our implementation of subroutine threading.

The choice of optimal dispatch technique depends on the hardware platform, because dispatch is highly dependent on micro-architectural features. On earlier hardware, *call* and *return* were both expensive and hence subroutine threading required two costly branches, versus one in the case of direct threading. Rodriguez [17] presents the tradeoffs for various dispatch types on several 8 and 16-bit CPUs. For example, he finds direct threading is faster than subroutine threading on a 6809 CPU, because the JSR and RET instruction require extra cycles to push and pop the return address stack. On the other hand, Curley found subroutine threading faster on the 68000 [3]. On modern hardware the cost of the *call* and *return* is much lower, due to return branch prediction hardware, while the cost of direct threading has increased due to misprediction. In Section 5 we demonstrate this effect on several modern CPUs.

Superinstructions reduce the number of dispatches. Consider the code to add a constant integer to a variable. This may require loading the variable onto the stack, loading the constant, adding, and storing back to the variable. VM designers can instead extend the virtual instruction set with a single superinstruction that performs the work of all four instructions. This technique is limited, however, because the virtual instruction encoding (often one byte per opcode) may allow only a limited number of instructions, and the number of desirable superinstructions grows exponentially in the number of subsumed atomic instructions. Furthermore, the optimal superinstruction set may change based on the workload. One approach uses profile-feedback to select and create the superinstructions statically (when the interpreter is compiled [8]).

Piumarta [15] presents *selective inlining*. It constructs superinstructions when the virtual program is loaded. They are created in a relatively portable way, by memcpy'ing the native code in the bodies, again using GNU C labels-as-values. This technique was first documented earlier [19], but Piumarta's independent discovery inspired many other projects to exploit selective inlining. Like us, he applied his optimization to OCaml, and reports significant speedup on several microbenchmarks. As we discuss in Section 5.4, our technique is separate from, but supports and indeed facilitates, inlining optimizations.

Only certain classes of opcode bodies can be relocated using memcpy alone—the body must contain no pc-relative instructions (typically this excludes C function calls). Selective inlining requires that the superinstruction starts at a virtual basic block, and ends at or before the end of the block. Ertl's *dynamic superinstructions* [6] also use memcpy, but are applied to effect a simple native compilation by inlining bodies for nearly every virtual instruction. Ertl shows how to avoid the virtual basic block constraints, so dispatch to interpreter code is only required for virtual branches and un-relocatable bodies. Catenation [24] patches Sparc native code so that all implementations can be moved, specializes operands, and converts virtual branches to native, thereby eliminating the virtual program counter.

Replication—creating multiple copies of the opcode body—decreases the number of contexts in which it is executed, and hence increases the chances of successfully predicting the successor [6]. Replication implemented by inlining opcode bodies reduces the number of dispatches, and therefore, the average dispatch overhead [15]. In the extreme, one could create a copy for each instruction, elimi-

nating misprediction entirely. This technique results in significant code growth, which may [24] or may not [6] cause cache misses.

In summary, misprediction of the indirect branches used by a direct threaded interpreter to dispatch virtual instructions limits its performance on modern CPUs because of the context problem. We have described several recent dispatch optimization techniques. Some of the techniques improve performance of each dispatch by reducing the number of contexts in which a body is executed. Others reduce the number of dispatches, possibly to zero.

Dynamo [2] is a system for trace-based runtime optimization of arbitrary programs. Its optimizations include replacing indirect branches with guarded linear control flow. One would expect this to be highly applicable to threaded interpreters. Sullivan et al. [22] applied Dynamo to a Java VM, but found it faired poorly. This was due to the context problem—it could not distinguish between the different runtime contexts of a bytecode body. The solution was to detect traces using a `<pc,vPC>` tuple, instead of only `pc`. Our technique, while simpler, accomplishes the same thing. In the following section we will describe how we address the context problem directly, by devirtualizing the interpreter's control flow and thus exposing virtual execution to native branch prediction resources.

4 Design and Implementation

Direct-threaded interpreters are known to have very poor branch prediction properties, however, they are also known to have a small cache footprint (for small to medium sized opcode bodies) [18]. Since both branches and cache misses are major pipeline hazards, we would like to retain the good cache behavior of direct-threaded interpreters while improving the branch behavior. The preceding section describes various techniques for improving branch prediction by replicating entire bodies. The effect of these techniques is to trade instruction cache size for better branch prediction. Ertl [6] claims that for Forth, with small opcode bodies, code growth occurs but does not cause cache-related stalls. Vitale and Abdelrahman [24] find that, with larger opcode bodies, code growth from replication does induce cache-misses. We believe it is best to avoid growing code if possible. We introduce a new technique which minimally affects code size and produces dramatically fewer branch mispredictions than either direct threading or direct threading with inlining.

In this section we motivate our design in terms of aligning virtual machine context with physical machine context, and outline our implementation.

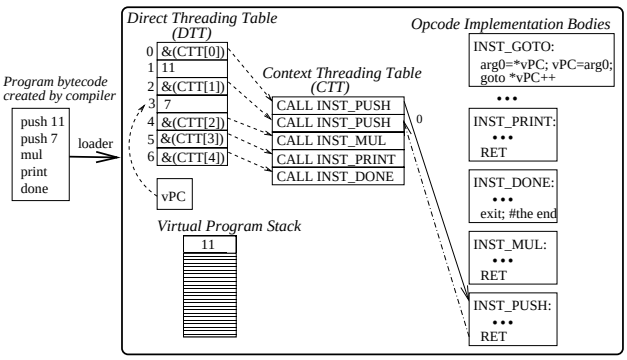


Figure 3. Subroutine Threaded VM Interpreter

4.1 Understanding Branches

To motivate our design, first note that the virtual program may contain all the usual types of control flow: conditional and unconditional branches, indirect branches, and calls and returns. We must also consider the dispatch of straight-line virtual instructions. For direct-threaded interpreters, sequential (virtual) execution is just as expensive as handling control transfers, since *all* virtual instructions are dispatched with an indirect branch. Second, note that the dynamic execution path of the virtual program will contain patterns (loops, for example) that are similar in nature to the patterns found when executing native code. These control flow patterns originate in the algorithm that the virtual program implements, whether it is interpreted or compiled.

Finally, note that modern microprocessors have considerable resources devoted to identifying these patterns in native code, and exploiting them to predict branches. In fact, the hardware provides different types of predictors to support different types of native branches. Unfortunately, direct threading uses only indirect branches and, due to the context problem, the patterns that exist in the virtual program are effectively hidden from the microprocessor.

The fundamental goal of our approach is to expose these virtual control flow patterns to the hardware, such that the physical execution path matches the virtual execution path. To achieve this goal, we exploit the different types of hardware prediction resources to handle the different types of virtual control flow transfers. In Section 4.2 we show how to replace straight-line dispatch with subroutine threading. In Section 4.3 we show how to inline conditional and indirect jumps and in Section 4.4 we discuss handling virtual calls and returns with native calls and returns. We strive to maintain the property that the virtual program counter is precisely correlated with the physical program counter and in fact, with our technique there is a one-to-one mapping between them at control flow points.

4.2 Handling Linear Dispatch

The dispatch of straight-line virtual instructions is the largest single source of branches when executing an interpreter. Any technique that hopes to improve branch prediction accuracy must thus address dispatch. The obvious solution is inlining, as it eliminates the dispatch entirely for straight-line sequences of virtual instructions. Inlining also has other benefits, such as enabling optimizations across the implementations of multiple virtual instructions. The increase in code size caused by aggressive inlining, however, has the potential to overwhelm the benefits with the cost of increased instruction cache misses.

Rather than eliminate dispatch, we propose an alternative organization for the interpreter in which native call and return instructions are used. Conceptually, this approach is elegant because subroutines are a natural unit of abstraction to express the implementations of virtual instructions.

Figure 3 illustrates our implementation of subroutine threading, using the same example program as Figure 1. In this case, we show the state of the virtual machine *after* the first virtual instruction has been executed (note that the virtual program stack now contains the value "11"). We add a new structure to the interpreter architecture, called the *Context Threading Table* (CTT), which contains a sequence of native *call* instructions. Each native *call* dispatches the body for its virtual instruction. We use the term *Context Threading*, because the hardware address of each call instruction in the CTT provides execution context to the hardware, most importantly, to the branch predictors. Each non-branching opcode body now ends with a native *return* instruction, while opcodes that modify the virtual control flow end with an indirect jump, as in direct-threading. The Direct Threading Table (DTT) is still necessary to store immediate virtual operands, and to correctly resolve virtual control transfer instructions. In direct threading, entries in the DTT point to opcode bodies, whereas in subroutine threading they refer to call sites in the CTT.

It seems counterintuitive to improve dispatch performance by calling each body. It is not obvious whether a call to a constant target is more or less expensive to execute than an indirect jump, but that is not the issue. Modern microprocessors contain specialized hardware to improve the performance of *call* and *return*— specifically, a return address stack that predicts the destination of the return to be the instruction following the corresponding call. Although the cost of subroutine threading is two control transfers, versus one for direct threading, this cost is outweighed by the benefit of eliminating a large source of unpredictable branches.

4.3 Handling Virtual Branches

Subroutine threading handles the branches that are induced by the dispatch of straight-line virtual instructions, however, the actual control flow of the virtual program is still hidden from the hardware. That is, bodies of opcodes that affect the virtual control flow still have no context. There are two problems, one relating to shared indirect branch prediction resources, and one relating to a lack of history context for conditional branch prediction resources.

Consider the implementation of `INST_GOTO` in Figure 3. Even for this simple unconditional virtual branch, prediction is problematic, because *all* `INST_GOTO` instructions in the virtual program share a single indirect branch instruction (and hence have a single prediction context). Conditional virtual branches have the same problem. A simple solution is to generate replicas of the indirect branch instruction in the CTT immediately following the call to the branching opcode body. Branching opcode bodies now end with native return, which transfers control to the replicated indirect branch in the CTT. As a consequence, each virtual branch instruction now has its own hardware context. We refer to this technique as *branch replication*.

Branch replication is attractive because it is simple, and produces the desired context with a minimum of replicated instructions. However, it has a number of drawbacks. First, for branching opcodes, we execute three hardware control transfers (a call to the body, a return, and the actual branch), which is an unnecessary overhead. Second, we still use the overly general indirect branch instruction, even in cases like `INST_GOTO` where we would prefer a simpler direct native branch. Third, by only replicating the dispatch part of the virtual instruction, we do not take full advantage of the conditional branch predictor resources provided by the hardware. Due to these limitations, we only use branch replication for indirect virtual branches and exceptions.

For all other branches we fully inline the bodies of virtual branch instructions into the CTT. We refer to this as *branch inlining*. In the process of inlining, we convert indirect branches into direct branches, where possible. We thus reduce pressure on the BTB, and instead exploit the conditional branch predictors. In particular, the virtual conditional branches now appear as real conditional branches to the hardware. The primary cost of branch inlining is increased code size, but this is modest because virtual branch instructions are simple and have small bodies. For instance, on the Pentium IV, most branch instructions can be inlined with no more than 10 words of additional space. Figure 4 shows an example of inlining the `INST_GOTO` branch instruction. The figure also illustrates how we handle virtual call/return control flow, described next.

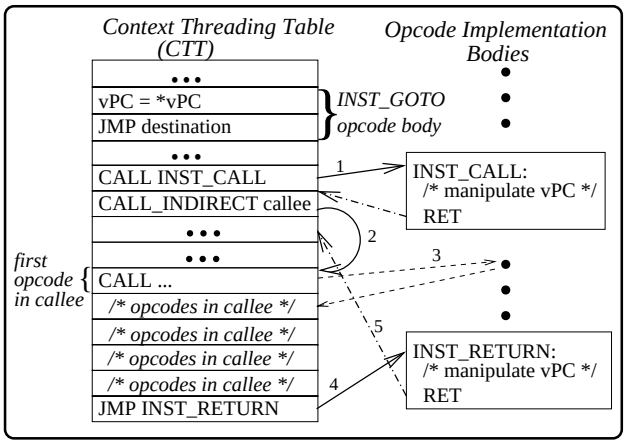


Figure 4. Context Threaded VM Interpreter: Branch and Return Inlining

4.4 Handling Virtual Call and Return

The only significant source of control transfers that remain in the virtual program are virtual calls and returns. For successful branch prediction, the real problem is not the virtual call, but rather the virtual return, because one virtual return may go back to multiple call sites. As noted previously, the hardware already has an elegant solution to this problem for native code in the form of the return address stack. We need only to deploy this resource to predict virtual returns.

We describe our solution with reference to Figure 4. The virtual call body should effect a transfer of control to the start of the callee. We begin at a virtual call instruction (see arrow labeled "1"). The virtual call body simply sets the `vPC` to refer to the virtual callee and executes a native *return* to the next CTT location. Similar to branch replication, we insert a new native *call indirect* instruction at this point in the CTT to transfer control to the start of the callee (arrow "2"). This *call indirect* causes the next location in the CTT to be pushed onto the hardware's return address stack. The first instruction of the callee is then dispatched (arrow "3"). At the end of the callee, we modify the virtual return instruction as follows. In the CTT, we emit a native direct branch to dispatch the body of the virtual return (arrow "4".) Unlike using a native *call* for this dispatch, the direct branch avoids perturbing the return address stack. We modify the body of the virtual return to end with a native *return* instruction, which now transfers control all the way back to the instruction following the original virtual call (arrow "5".) We refer to this technique as *apply/return inlining*[1].

With this final step, we have a complete technique that aligns all virtual program control flow with the corresponding native flow. There are however, some practical challenges to implementing our design for apply/return inlining. First, one must take care to match the hardware stack against the virtual program stack. For instance, in OCaml, exceptions unwind the virtual machine stack; the hardware stack must be unwound in a corresponding manner. Second, some run-time environments are extremely sensitive to hardware stack manipulations, since they use or modify the machine stack pointer for their own purposes (such as handling signals). In such cases, it is possible to create a separate stack structure and swap between the two at virtual call and return points. This approach would introduce significant overhead, and is only justified if apply/return inlining provides a substantial performance benefit.

Having described our design and its general implementation, we now evaluate its effectiveness on real interpreters.

[1] "apply" is the name of the (generalized) function call opcode in OCaml

5 Experimental Evaluation

In this section, we evaluate the performance of context threading and compare it to direct threading and direct-threaded selective inlining. Context threading combines subroutine threading, branch inlining and apply/return inlining. We evaluate the contribution of each of these techniques to the overall impact of context threading using two virtual machines and three microprocessor architectures. We begin by describing our experimental setup in Section 5.1. We then investigate how effectively our techniques address pipeline branch hazards in Section 5.2, and the overall effect on execution time in Section 5.3. Finally, Section 5.4 demonstrates that context threading is complementary to inlining resulting in a portable, relatively simple, technique that provides performance comparable to or better than SableVM's implementation of selective inlining.

5.1 Virtual Machines, Benchmarks and Platforms

OCaml We chose OCaml as representative of a class of efficient, stack-based interpreters that use direct-threaded dispatch. The bytecode bodies of the interpreter are very efficient, and have been hand-tuned, including register allocation. The implementation of the OCaml interpreter is clean and easy to modify.

SableVM SableVM is a Java Virtual Machine built for quick interpretation, implementing lazy method loading and a novel bi-directional virtual function lookup table. Hardware signals are used to handle exceptions. Most importantly for our purposes, SableVM already implements multiple dispatch mechanisms, including switch, direct threading, and selective inlining (which SableVM calls *inline threading*) [9]. The support for multiple dispatch mechanisms makes it easy to add context threading, and allows us

Table 1. Description of OCaml benchmarks

Benchmark	Description	Pentium IV Time (TSC*10^8)	Pentium IV Branch Mispredicts (MPT*10^6)	PowerPC 7410 Time (Cycles*10^8)	PowerPC 7410 Branch Stalls (Cycles*10^6)	PPC970 Elapsed Time (sec)	Lines of Source Code
boyer	Boyer theorem prover	3.34	7.21	1.8	43.9	0.18	903
fft	Fast Fourier transform	31.9	52.0	18.1	506	1.43	187
fib	Fibonacci by recursion	2.12	3.03	2.0	64.7	0.19	23
genlex	A lexer generator	1.90	3.62	1.6	27.1	0.11	2682
kb	A knowledge base program	17.9	42.9	9.5	283	0.96	611
nucleic	nucleic acid's structure	14.3	19.9	95.2	2660	6.24	3231
quicksort	Quicksort	9.94	20.1	7.2	264	0.70	91
sieve	Sieve of Eratosthenes	3.04	1.90	2.7	39.0	0.16	55
soli	A classic peg game	7.00	16.2	4.0	158	0.47	110
takc	Takeuchi function (curried)	4.25	7.66	3.3	114	0.33	22
taku	Takeuchi function (tuplified)	7.24	15.7	5.1	183	0.52	21

Table 2. Description of SpecJVM benchmarks

Benchmark	Description	Pentium IV Time (TSC*10^{11})	Pentium IV Branch Mispredicts (MPT*10^9)	PowerPC 7410 Time (Cycles*10^{10})	PowerPC 7410 Branch Stalls (Cycles*10^8)	PPC970 Elapsed Time (sec)
compress	Modified Lempel-Ziv compression	4.48	7.13	17.0	493	127.7
db	performs multiple database functions	1.96	2.05	7.5	240	65.1
jack	A Java parser generator	0.71	0.65	2.7	67	18.9
javac	the Java compiler from the JDK 1.0.2	1.59	1.43	6.1	160	44.7
jess	Java Expert Shell System	1.04	1.12	4.2	110	29.8
mpegaudio	decompresses MPEG Layer-3 audio files	3.72	5.70	14.0	460	106.0
mtrt	two thread variant of raytrace	1.06	1.04	5.3	120	26.8
raytrace	a raytracer rendering	1.00	1.03	5.2	120	31.2
scimark	performs FFT SOR and LU, 'large'	4.40	6.32	18.0	690	118.1
soot	java bytecode to bytecode optimizer	1.09	1.05	2.7	71	35.5

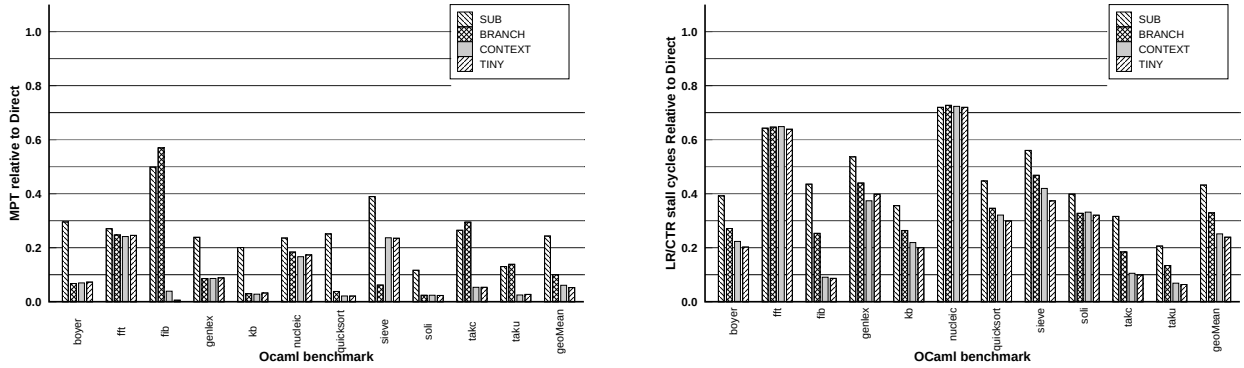
to compare it against a selective inlining implementation, which we believe is a more complicated technique.

OCaml Benchmarks The benchmarks in Table 1 constitute the complete standard OCaml benchmark suite[2]. Boyer, kb, quicksort and sieve are mostly integer processing, while nucleic and fft are mostly floating point benchmarks. Soli is an exhaustive search algorithm that solves a solitaire peg game. Fib, taku, and takc are tiny, highly-recursive programs which calculate integer values. These three benchmarks are unusual because they contain very few distinct virtual instructions, and often contain only one instance of each. These features have two important consequences. First, the indirect branch in direct-threaded dispatch is relatively predictable. Second, even minor changes can have dramatic effects (both positive and negative) because so few instructions contribute to the behavior.

[2] `ftp://ftp.inria.fr/INRIA/Projects/cristal/Xavier.Leroy/benchmarks/objcaml.tar.gz`

SableVM Benchmarks SableVM experiments were run on the complete SPECjvm98 [20] suite (compress, db, mpegaudio, raytrace, mtrt, jack, jess and javac), one large object oriented application (soot [23]) and one scientific application (scimark [16]). Table 2 summarizes the key characteristics of these benchmarks.

Pentium IV Measurements The Pentium IV (P4) processor aggressively dispatches instructions based on branch predictions. As discussed in Section 2, the taken indirect branches used for direct-threaded dispatch are often mispredicted due to the lack of context. Ideally, we would measure the mispredict penalty for these branches to see their effect on execution time, but the P4 does not have a counter for this purpose. Instead, we count the number of *mispredicted taken branches* (MPT) to show how effectively



(a) Pentium IV MPT

(b) PPC7410 lr/ctr stalls

Figure 5. OCaml Pipeline Hazards Relative to Direct Threading

context threading improves branch prediction. We measure time on the P4 with the cycle-accurate *time stamp counter* (TSC) register. We count both MPT and TSC events using our own Linux kernel module, which collects complete data for the multithreaded Java benchmarks[3].

PowerPC Measurements We need to characterize the cost of branches differently on the PowerPC than on the P4, as these processors do not typically speculate on indirect branches[4]. Instead, split branches are used (as shown in Figure 2(b)) and the PPC stalls in the branch unit until the branch destination is known. Hence, we would like to count the number of cycles stalled due to link and count register dependencies. Fortunately, the older PPC7410 CPU has a counter (counter 15, "stall on LR/CTR dependency") that provides exactly this information [14]. On the PPC7410, we also use the hardware counters to obtain overall execution times in terms of clock cycles. We expect that the branch stall penalty should be larger on more deeply-pipelined CPUs like the PPC970, however, we cannot directly count these stall cycles on this processor. Instead, we report only elapsed execution time for the PPC970.

Interpreting the data In presenting our results, we normalize all experiments to the direct threading case, since it is the baseline state-of-the art dispatch technique. We give the absolute execution times and branching characteristics for each benchmark and platform using direct threading in Tables 1 and 2. Bar graphs in the following sections show the contributions of each component of our technique: subroutine threading only (labeled `SUB`); subroutine threading plus branch inlining and branch replication for exceptions and indirect branches (labeled `BRANCH`); and our complete context threading implementation which includes apply/return inlining (labeled `CONTEXT`). We include bars for selective inlining in SableVM (labeled `SELECT`) and our own simple inlining technique (labeled `TINY`) to facilitate comparisons, although inlining results are not discussed until Section 5.4. We do not show a bar for direct threading because it would have height 1.0, by definition.

[3] MPT events are counted with performance counter 8 by setting the P4 CCCR to 0x0003b000 and the ESCR to value 0xc001004 [11]

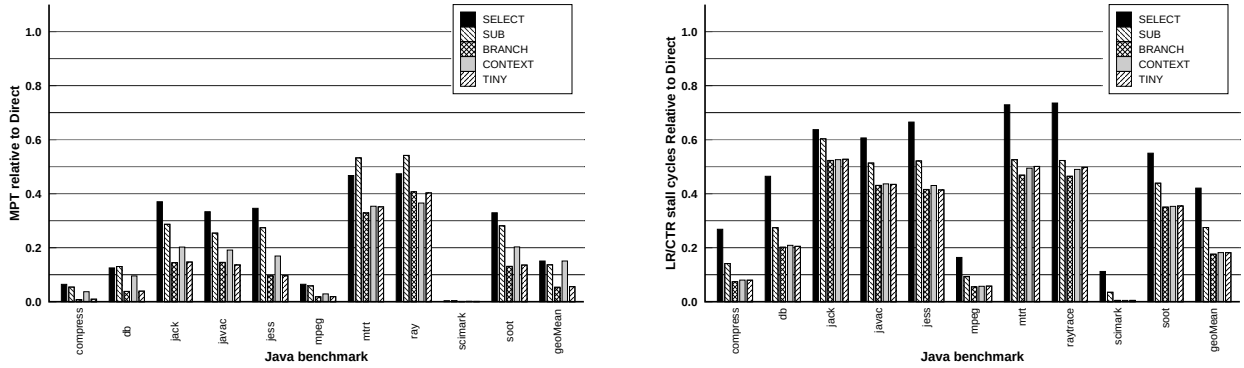
[4] A "hint bit" can be used to encourage speculation in later models like the PPC970 but it is not used by default.

5.2 Effect on Pipeline Branch Hazards

Context threading was designed to align virtual program state with physical machine state to improve branch prediction and reduce pipeline branch hazards. We begin our evaluation by examining how well we have met this goal.

Figure 5 reports the extent to which context threading reduces pipeline branch hazards for the OCaml benchmarks, while Figure 6 reports these results for the Java benchmarks on SableVM. On the left of each Figure, the graphs labeled (a) present the results on the P4, where we count mispredicted taken branches (MPT). On the right, graphs labeled (b) present the effect on LR/CTR stall cycles on the PPC7410. The last cluster of each bar graph reports the geometric mean across all benchmarks.

Context threading eliminates most of the mispredicted taken branches (MPT) on the Pentium IV and LR/CTR stall cycles on the PPC7410, with similar overall effects for both interpreters. Examining Figures 5 and 6 reveals that subroutine threading has the single greatest impact, reducing MPT by an average of 75% for OCaml and 85% for SableVM on the P4, and reducing LR/CTR stalls by 60% and 75% on average for the PPC7410. This result matches our expectations because subroutine threading addresses the largest single source of unpredictable branches—the dispatch used



(a) Pentium IV MPT

(b) PPC7410 lr/ctr stalls

Figure 6. SableVM Pipeline Hazards Relative to Direct Threading

for all straight-line bytecodes. Branch inlining has the next largest effect, again as expected, since conditional branches are the most significant remaining pipeline hazard after applying subroutine threading. On the P4, branch inlining cuts the remaining MPTs by about 60%. On the PPC7410 branch inlining has a smaller, though still important effect, eliminating about 25% of the remaining LR/CTR stall cycles. A notable exception to the MPT trend occurs for the OCaml benchmarks fib, takc and taku. In these tiny, recursive benchmarks de-virtualizing the conditional branches hurts prediction by a small amount. As noted previously, even minor changes in the behavior of a single instruction can have a noticeable impact for these benchmarks.

Interestingly, the same OCaml benchmarks that are a challenge for branch inlining on the P4 also reap the greatest benefit from apply/return inlining, as shown in Figure 5(a). Due to the recursive nature of these benchmarks, their performance is dominated by the behavior of virtual calls and returns. Thus, mapping these operations to native calls and returns has an enormous impact. For sieve, on the P4, the result of apply/return inlining is an increase in MPT, while for the non-recursive OCaml benchmarks, the overall effect on both platforms is a small improvement.

For SableVM on the P4, however, apply/return inlining is restricted by the fact that SableVM uses the processor's `esp` register. Rather than implement a complicated stack switching technique as discussed in Section 4.4, we allow the virtual and machine stacks to become mis-aligned when SableVM manipulates the `esp` directly. This increases the overhead of our apply/return inlining implementation and reduces the effectiveness of the return address stack predictor, as can be seen in the bar labeled CONTEXT in Figure 6(a). On the PPC7410, the effect of apply/return inlining on LR/CTR stalls is very small for SableVM.

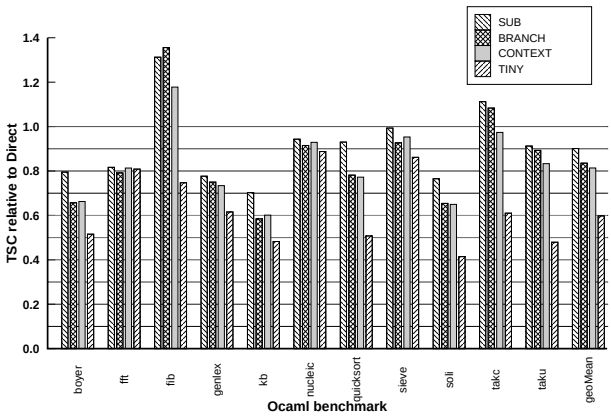
Having shown that our techniques can significantly reduce pipeline branch hazards, we now examine the impact of these reductions on overall execution time.

5.3 Performance

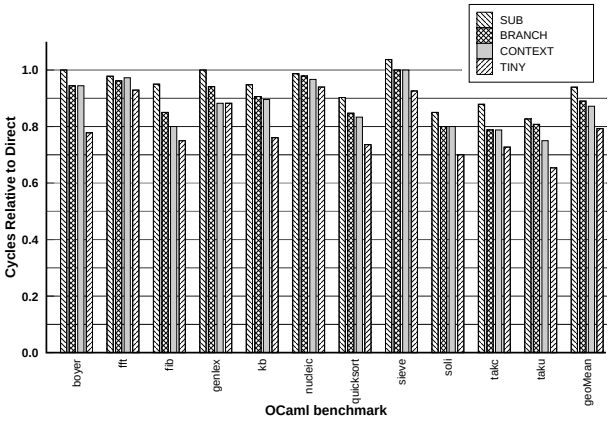
Context threading improves branch prediction, resulting in increased pipeline usage on both the P4 and the PPC. However, using a native *call/return* pair for each dispatch increases instruction overhead. In this section, we examine the net result of these two effects on overall execution time. As before, all data is reported relative to direct threading.

Figures 7 and 8 show results for the OCaml and SableVM benchmarks respectively. They are organized in the same way as the previous section, with P4 results on the left, labeled (a), and PPC7410 results on the right, labeled (b). Figure 9 reports the performance of OCaml and SableVM on the PPC970 CPU. The geometric means (rightmost cluster) in Figures 7, 8 and 9 show that context threading significantly outperforms direct threading on both virtual machines and on all three architectures. The geometric mean execution time of the Ocaml VM is about 19% lower for context threading than direct threading on P4, 9% lower on PPC7410, and 39% lower on the PPC970. For SableVM, context threading, compared with direct threading, runs about 17% faster on the PPC7410 and 26% faster on both the P4 and PPC970. Although we cannot measure the cost of LR/CTR stalls on the PPC970, the greater reductions in execution time are consistent with its more deeply-pipelined design (23 stages vs. 7 for the PPC7410).

Across interpreters and architectures, the effect of our techniques is clear. Subroutine threading has the single largest impact on elapsed time. Branch inlining has the next largest impact eliminating an additional 3–7% of the elapsed time. In general, the reductions in execution time track the reductions in branch hazards seen in Figures 5 and

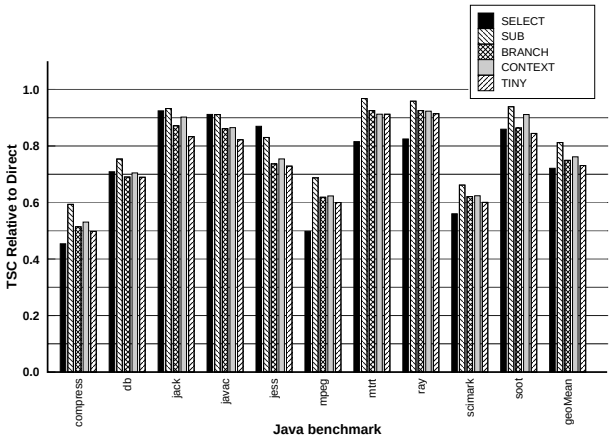


(a) Pentium IV TSC

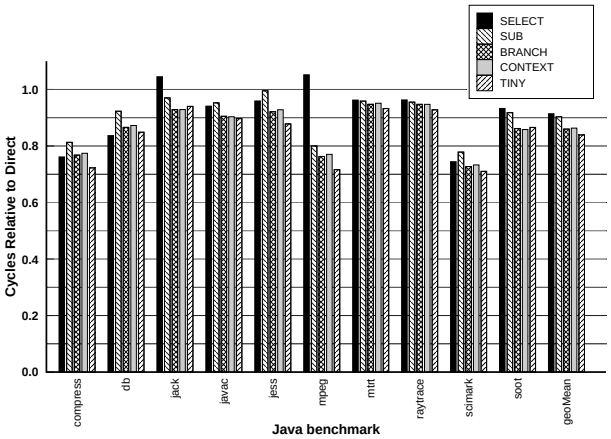


(b) PPC7410 cycles

Figure 7. OCaml Elapsed Time Relative to Direct Threading

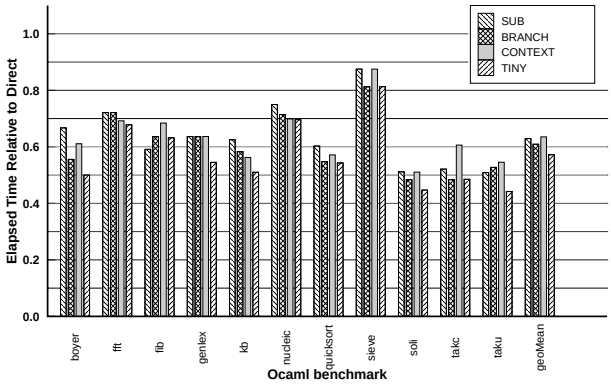


(a) Pentium IV TSC

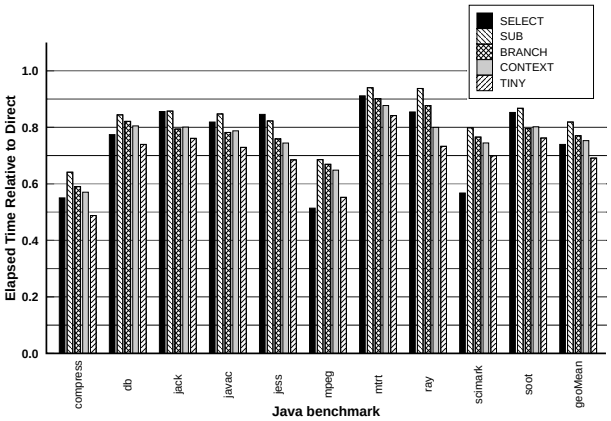


(b) PPC7410 cycles

Figure 8. SableVM Elapsed Time Relative to Direct Threading



(b) OCaml PPC970 elapsed (real) sec



(a) SableVM PPC970 elapsed (real) sec

Figure 9. PPC970 Elapsed Time Relative to Direct Threading

6. The instruction overheads of our dispatch technique are most evident in the OCaml benchmarks fib and takc on the P4 where the benefits of improved branch prediction (relative to direct threading) are minor. In these cases, the opcode bodies are very small and the extra instructions executed for dispatch are the dominant factor.

The effect of apply/return inlining on execution time is minimal overall, changing the geometric mean by only $\pm 1\%$ with no discernible pattern. Given the limited performance benefit and added complexity, a general implementation of apply/return inlining does not seem worthwhile. Ideally, one would like to detect heavy recursion automatically, and only perform apply/return inlining when needed. We conclude that, for general usage, subroutine threading plus branch inlining provides the best trade-off.

We now demonstrate that context-threaded dispatch is complementary to inlining techniques.

5.4 Inlining

Inlining techniques address the context problem by replicating bytecode bodies and removing dispatch code. This reduces both instructions executed as well as pipeline hazards. In this section we show that, although both selective inlining and our context threading technique reduce pipeline hazards, context threading is slower because of instruction overhead. We address this issue by comparing our own *tiny inlining* technique with selective inlining.

In Figures 6, 8 and 9(a) the bar labeled `SELECT` shows our measurements of Gagnon's selective inlining implementation for SableVM [9]. From these Figures, we see that selective inlining reduces both MPT and LR/CTR stalls significantly as compared to direct threading, but it is not as effective in this regard as subroutine threading alone. The larger reductions in pipeline hazards for context threading, however, do not necessarily translate into better performance over selective inlining. Figure 8(a) illustrates that SableVM's selective inlining beats context threading on the P4 by roughly 5%, whereas on the PPC7410 and the PPC970, both techniques have roughly the same effect on execution time, as shown in Figure 8(b) and Figure 9(a), respectively. These results show that reducing pipeline hazards caused by dispatch is not sufficient to match the performance of selective inlining. By eliminating some dispatch code, selective inlining can do the same real work with fewer instructions than context threading.

Context threading is only a dispatch technique, and can be easily combined with inlining strategies. To investigate the impact of dispatch instruction overhead and to demonstrate that context threading is complementary to inlining, we implemented *Tiny Inlining*, a simple heuristic that inlines all bodies with a length less than four times the length of our dispatch code. This eliminates the dispatch overhead surrounding the smallest bodies and, as calls in the CTT are replaced with comparably-sized bodies, tiny inlining ensures that the total code growth is minimal. In fact, the smallest inlined OCaml bodies on P4 were *smaller* than the length of a relative call instruction. Table 3 summarizes the effect of tiny inlining. On the P4, we come within 1% of SableVM's sophisticated selective inlining implementation. On PowerPC, we outperform SableVM by 7.8% for the PPC7410 and 4.8% for the PPC970.

Table 3. Selective Inlining vs Context+Tiny (SableVM)

Arch	Context (C)	Selective (S)	Tiny (T)	Δ (S-C)	Δ (S-T)
P4	0.762	0.721	0.731	−0.041	−0.010
PPC7410	0.863	0.914	0.839	0.051	0.075
PPC970	0.753	0.739	0.691	−0.014	0.048

The primary costs of direct-threaded interpretation are pipeline branch hazards, caused by the context problem. Context threading solves this problem by correctly deploying branch prediction resources, and as a result, outperforms direct threading by a wide margin. Once the pipelines are full, the secondary cost of executing dispatch instructions is significant. A suitable technique for addressing this overhead is inlining, and we have shown that context threading is compatible with inlining using the "tiny" heuristic. Even with this simple approach, context threading achieves performance equivalent to, or better than, selective inlining.

6 Current and Future Work

At the time of writing, we have extended our context threading technique with a general purpose framework which allows for the safe execution of arbitrary instrumentation code in between bytecodes. Within this framework, we have implemented bytecode logging to assist with debugging and several frequency and branch bias profilers. We have developed a lazy linking technique that allows us to dynamically add generated code segments. Using these tools, we currently identify hot basic blocks, then regenerate and link them into the program on the fly. We intend to use these capabilities to dynamically generate code for other interesting compilation units including loop bodies, traces, and whole methods.

7 Conclusions

Modern CPUs have deep pipelines which must be kept full for them to perform well. Filling these pipelines requires that the processor speculate on which instructions it

will be executing by predicting the direction or target of branching instructions. Direct threaded interpreters use indirect branches to dispatch bytecode bodies. On older processors this was efficient, but on modern processors these branches are pipeline hazards causing either stalls or flushes due to mispredictions, hurting performance.

Context threading improves performance by exposing the virtual program's control flow to the hardware, reducing pipeline hazards. For sequential bytecodes, we use subroutine threading, dispatching each bytecode with a relative call and predicting each successor with the return stack. We inline the bodies of virtual conditional instructions, exposing the virtual execution context to the hardware's conditional branch predictors. We also demonstrate a technique for improving prediction of virtual returns. We have shown that our techniques eliminate a significant number of pipeline branch hazards. We reduce mean branch mispredictions by 95% on the P4 and reduce mean LR/CTR branch unit stall cycles by between 76% and 82% on the PowerPC 7410. Relative to direct threading, context threading reduces mean elapsed time by 19 to 27% on the Pentium IV, 14% on the PowerPC 7410 and by 20 to 37% on the PowerPC 970.

Context threading performs better than direct threading but worse than direct-threaded inlining. We implemented a simple inlining scheme in which we inline only very small bytecodes. On the P4 we come within 1% of SableVM's sophisticated selective inlining implementation. On the PPC970 we outperform SableVM's implementation of selective inlining by 4.8%.

Context threading is easy to implement, and provides a significant performance advantage over direct threading. It addresses the main obstacle to high performance virtual instruction dispatch by exposing the virtual program's control flow to the hardware's control flow predictors. Furthermore, the context threading technique is simple, so the code remains flexible and supports other optimizations such as inlining and code specialization. We thus conclude that context threading is a better technique for baseline interpretation and may be an attractive environment for dynamic optimization.

References

[1] The Java hotspot virtual machine, v1.4.1, technical white paper. 2002.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[3] C. Curley. Life in the FastForth lane. *Forth Dimensions*, 14(4), January-February 1993.

[4] C. Curley. Optimizing in a BSR/JSR threaded forth. *Forth Dimensions*, 14(5), March-April 1993.

[5] M. A. Ertl and D. Gregg. The behavior of efficient virtual machine interpreters on modern architectures. *Lecture Notes in Computer Science*, 2150, 2001.

[6] M. A. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proc. ACM SIGPLAN 2003 PLDI*, 2003.

[7] M. A. Ertl and D. Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 2003.

[8] M. A. Ertl, D. Gregg, A. Krall, and B. Paysan. VMgen — a generator of efficient virtual machine interpreters. *Software Practice and Experience*, 32:265–294, 2002.

[9] E. Gagnon and L. J. Hendren. In *Proc. of 12th Intl. Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 170–184. Springer, 2003.

[10] G. Hinton, D. Sagar, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Q1, 2001.

[11] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*. 2004.

[12] P. M. Kogge. An architectural trail to threaded- code systems. *IEEE Computer*, 15(3), March 1982.

[13] Motorola Corporation. *MPC7410 RISC Microprocessor Programmer's Manual*. 2002.

[14] Motorola Corporation. *MPC7410/MPC7400 RISC Microprocessor User's Manual, Rev. 1*. 2002.

[15] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN Proc. PLDI*, pages 291–300, 1998.

[16] R. Pozo and B. Miller. *SciMark: a numerical benchmark for Java and C/C++*., 1998.

[17] B. Rodriguez. Benchmarks and case studies of forth kernels. *The Computer Journal*, 60, 1993.

[18] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy. The structure and performance of interpreters. In *Proc. ASPLOS 7*, pages 150–159, October 1996.

[19] M. Rossi and K. Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. Technical Report TKO-C79, Helsinki University Faculty of Information Technology, May 1996.

[20] SPEC jvm98 benchmarks, 1998.

[21] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java just-in-time compiler. *IBM Systems Journals, Java Performance Issue*, 39(1), February 2000.

[22] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic native optimization of interpreters. In *Proc. of the Workshop on Interpreters, Virtual Machines and Emulators*, 2003.

[23] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[24] B. Vitale and T. S. Abdelrahman. Catenation and operand specialization for Tcl VM performance. In *Proc. 2nd IVME*, pages 42–50, 2004.