Mixed Mode Execution with Context Threading

Mathew Zaleski
Department of Computer Science
University of Toronto
matz@cs.toronto.edu

Marc Berndl
Department of Computer Science
University of Toronto
berndl@cs.toronto.edu

Angela Demke Brown
Department of Computer Science
University of Toronto
demke@cs.toronto.edu

Abstract

Interpreters are widely used to implement portable language runtime environments. Programs written in these languages may benefit from performance beyond that obtainable by optimizing interpretation alone. A modern high-performance mixed-mode virtual machine (VM) includes a method-based Just In Time (JIT) compiler. A method-based JIT, however, requires the up-front development of a complex compilation infrastructure before any performance benefits are realized.

Ideally, the architecture for a mixed-mode VM could detect and execute a variety of shapes of hot regions of a virtual program. Our VM architecture is based on context threading. It supports powerful, efficient instrumentation and a simple framework for dynamic code generation. It has the potential to directly support a full spectrum of mixed-mode execution: from interpreted bytecode bodies, to specialized bytecode generated at runtime, to traces, to compiled methods. Further, it provides the necessary tools to detect these regions at runtime.

We extended two VMs, SableVM and the OCaml interpreter with our infrastructure on both the P4 and PPC. To demonstrate the power and flexibility of our infrastructure we compare the selection and dispatch effectiveness for three common region shapes: whole methods, partial methods, and SPECL traces. We report results for a preliminary version of our code generator which compiles a region into a sequence of direct calls to bytecode bodies.

1 Introduction

There is a significant gap between the performance of direct threaded interpretation and optimized, mixed-mode, Just in Time (JIT) compiled native code. Nevertheless, there are good reasons why the users of relatively few interpreted languages enjoy the benefits offered by a modern optimizing JIT. A JIT translates and optimizes all the bytecodes of a method, or, in many cases, an inlined method nest, into native code. This means that the back end of a JIT typically cannot be deployed until it supports essentially all the features of the language it translates. As a result, adding a JIT to an interpreter requires a large up-front investment in compilation infrastructure before any performance benefits are seen by users.

We claim what is needed is a virtual machine (VM) architecture that enables developers to gradually deploy specific performance improvements as they incrementally invest in infrastructure to support optimization and native code generation. It should execute in mixed-mode and be able to generate native code for variously shaped and sized regions: from single bytecode bodies, to basic blocks, to traces, to entire methods. Variably sized and shaped regions allow VM developers to focus on regions of virtual programs that manifest performance issues first. It should be able to intersperse the execution of generated code with the dispatch of bytecode bodies. This frees VM developers from needlessly generating code for language features that are complex or unprofitable to compile. Finally, it should be reasonably compatible with existing direct threaded interpreters to facilitate retro-fitting existing implementations.

We believe that this challenge breaks down naturally into two main concerns: performance and function. It must be possible to generate high quality code for a region of virtual program regardless of its shape. Since much research exists on how to dynamically optimize methods we instead focus on the problem of identifying, translating, linking and executing variously shaped regions of a virtual program.

In this paper, we introduce a new way of organizing a mixed-mode VM. We show how variably shaped regions can be identified and executed. Section 2 introduces the basics of interpretation. In addition, we briefly describe Context Threading (CT), a set of our own extensions and performance improvements to interpretation [5]. Much of the approach we describe below was informed by our experiences as we built CT.

Section 3 describes related ways of optimizing interpreter performance. Section 4 describes how we have modified two direct threaded VMs (SableVM 1.18 and OCaml 3.08) on two platforms (P4 and PPC) to select and translate hot regions of the virtual program. Our system selects various shapes of virtual program including: basic blocks, cached basic blocks[13], traces [3], methods and partial methods [23]. Once hot regions of the virtual program are identified, we generate code for them using CT, which generates a sequence of call instructions to bytecode bodies. Consequently, the overall performance of our system is similar to subroutine threading and is in no way competitive with an aggressive JIT. We are actively working on an optimizing code generator but are not yet in a position to report performance results. In Section 5 we report how the various shapes vary in their potential to maximize the time spent executing translated code relative to the amount of code translated. We report statistics reminiscent of those in [6].

2 Background

Interpreters execute a virtual program by dispatching its virtual instructions in sequence. The current instruction is indicated by a virtual program counter, or `vPC`. Each virtual instruction consists of an opcode and zero or more operands. Since the work of dispatch is a significant factor in interpreter performance, much effort has been made to optimize it. In this section we briefly review two traditional interpreter dispatch techniques (subroutine threaded code and direct threaded code) and our recently-introduced dispatch technique, context threaded code.

2.1 Direct-threaded dispatch

As shown on the left of Figure 1, a direct-threaded interpreter represents the instructions in a virtual program as a *list of addresses*. We refer to the block of implementation code as the *opcode body*, or simply *body*. Each address points to an opcode body. We refer to this list as the *Direct Threading Table*, or DTT. Virtual instruction operands are also stored in this list, immediately after the virtual opcode address. The `vPC` points into the DTT to indicate the next virtual 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 expressed in GNU C using its computed goto extension as `goto *vPC++` at the end of each opcode body. Direct-threaded dispatch is faster than a simple switch-based dispatch because it requires fewer control transfers, and with the GCC extensions, it is as portable as GNU C itself. However, although direct-threading requires only three instructions, it can still consume many more machine cycles, because the indirect branch may be poorly predicted and a pipeline hazard on modern CPUs [10].

2.2 Subroutine-threaded dispatch

Subroutine threading is described in the Forth literature, sometimes with other names [8]. It is often described as interesting largely as a simple first step towards generating native code [16, 9], which is amenable to local transformations such as inlining and peephole optimization. Subroutine threading eschews an interpreter dispatch loop, instead expressing the virtual program as a series of native machine `call` instructions to the virtual opcode bodies. Each body must end with a native `return` instruction. The literature is not always clear on the representation of virtual operands, nor handling of virtual branches in the absence of a traditional dispatch loop.

Figure 1(b) illustrates our implementation of subroutine threading. In this case, we show the state of the virtual machine *after* the first virtual instruction has been executed. We have added a new structure to the interpreter, called the *Context Threading Table* (CTT), which contains a sequence of native `call` instructions. Each non-branching opcode body 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 operands to the virtual instructions, and to correctly resolve virtual control transfer instructions. Whereas under direct threading entries in the DTT point to bodies, under CT

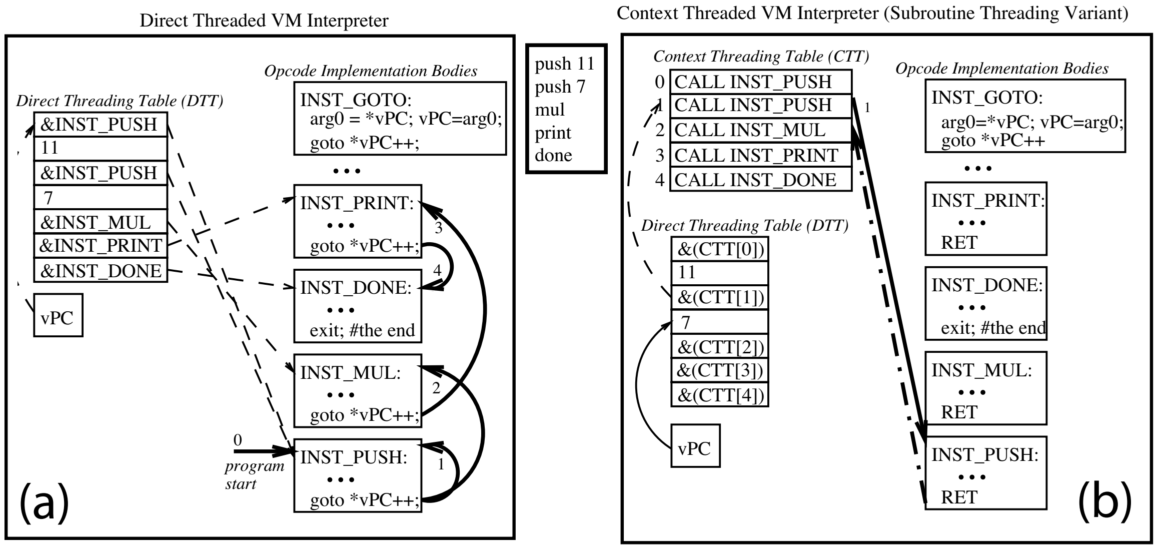

Figure 1: Direct threaded dispatch (a) compared to subroutine threaded dispatch (b) showing how Direct Threading Table stores immediate arguments to virtual instructions and points to the implementation of each virtual instruction.

they instead refer to call sites.

The cost of subroutine threading is a longer dispatch path (as compared to direct-threading), however calls and returns are predicted reliably by hardware on modern CPUs and the benefit of eliminating a large source of unpredictable branches outweighs this cost [5]. Direct threading and subroutine threading present similar implementation difficulties, however, subroutine threading requires the generation of three native instructions (`call`, `return` and `jump`) making it somewhat less portable.

2.3 Context-threaded dispatch

Subroutine threading eliminates mispredicted branches that are caused by the dispatch of straightline virtual instructions, however, the actual control flow of the virtual program is still seen by the hardware as unpredictable indirect branches. The issue is that the bodies of opcodes that affect the virtual control flow still have no context. The basic solution is to generate the branching code directly into the CTT (rather than dispatching the opcode body), such that each virtual branch is represented by a distinct hardware-visible branch instruction. This allows the microprocessor to deploy more of its branch predictor resources, further reducing mispredictions and improving performance.

While designing Context Threading (CT) we were struck with the relative ease with which it was possible to cleanly integrate the generation and dispatch of small regions of code into the interpreters. In retrospect we believe this was at least partially due to the fortuitous combination of subroutine dispatch and the direct threading table (DTT). The combination is a good one for three main reasons. First, the availability of the direct threading table (DTT) and the virtual PC facilitates *soft linking*, whereby newly generated native code can be dispatched by branching indirectly via entries in the DTT. Second, the fact that bytecode bodies are implemented as callable units makes generating code easier, as optimized generated code can be interspersed with calls to existing bytecode bodies. Third, the call/return structure of subroutine threading returns control to generated code between the execution of each bytecode body. Normally one would expect the provision of convenient interposition hooks to cost performance, but in fact it is a free side-effect of subroutine threaded dispatch. We illustrate the way these factors work well together by describing how they simplified some aspects of our implementation of CT.

2.3.1 *Generating forward branch code*

Our implementation of CT uses soft linking and self modifying code to simplify the implementation of forward branches. When a CT interpreter loads a method we wish to generate a direct branch instruction into the CTT for each virtual relative branch instruction. However, a simple one-pass loader has no way of knowing the destination of a forward virtual branch. The obvious approach is to build a two-pass or slightly more elaborate one-pass loader that delays generating the forward branch until its destination is known. However, by the time the whole method is loaded the DTT slot corresponding to the destination of each forward branch will be correctly set. This suggests that generation of the actual forward branch might be even simpler if it is delayed until the virtual branch instruction is actually executed for the first time.

For these reasons we implemented forward branches in CT using self-modifying code instead. Our CT enabled loader initially loads a forward virtual branch as a direct call to a short trampoline. The trampoline is called the first time the virtual instruction is executed and hence can determine the proper destination of the forward branch by inspecting the DTT. The trampoline then rewrites the direct call instruction as a direct branch to reach the proper destination.

We describe these details because they illustrate a scenario where simple dynamic infrastructure enables static analysis (admittedly an almost trivial two-pass loader in this case) to be replaced with run-time instrumentation. It demonstrates how the combination of the DTT, an up-to-date `vPC` and code rewriting can be used to maintain the emerging flow graph of a virtual program.

2.3.2 *Instrumentation and Profiling*

When debugging and profiling our implementation of CT we found that a convenient way to instrument our interpreter was to generate a call to a C instrumentation function as part of the dispatch for every virtual instruction. Profiling hooks are passed the virtual address (DTT address) and physical address (CTT address) of the virtual instruction about to be dispatched. In addition each hook needs to learn the address of a profiling data struct specific to each virtual instruction. Typical systems use a hash table to associate each virtual instruction with the profiling data used to record its behavior. The problem with this approach is that the hash table lookup will likely cost more than the execution of many virtual instructions. Instead, we generate code to pass the address of the profiling structure as one of the parameters of the C instrumentation function. We expect (though we do not present data to prove it) that this approach results in instrumentation that is much faster than a hash table based implementation.

This experience informed the design of the approach used to collect profiling information in support of region selection as described in Section 4.

3 Related Work

Various techniques aim to optimize direct-threaded interpretation by creating specialized versions of bytecode bodies. In this section, we discuss several dispatch optimizations, including replication, superinstruction formation, selective inlining, catenation, and trace-based optimization. Our infrastructure facilitates these techniques by providing both a means to profile interpreted code to detect regions of interest, and a means to create specialized versions of these regions.

Replication [11, 13] creates multiple copies of the same body. In so doing the indirect branches that dispatch to the next instruction are provided with their own hardware context and are thus better predicted. *Superinstructions* [15, 12] combine common sequences of bodies, eliminating the dispatch between them and generating a new dispatch with the combined effect on the `vPC` at the end. Piumarta and Riccardi's [13] *selective inlining* constructs superinstructions at load time. These are created in a relatively portable way, by `memcpy`'ing the native code in the bodies, using GNU C `goto` extensions. Our technique enables a runtime variation of selective inlining whereby we construct superinstructions only after the linear blocks they are derived from have run, as described in Section 4.3.2.

Catenation [22] inlines all bytecode bodies, propagates immediate arguments, relocates calls and replaces virtual branches with natives ones. The result is the elimination of the virtual program counter and all dispatch. Vitale and Abdelrahman report that inlining all Tcl bytecodes (which are relatively large) caused instruction cache performance problems. Curley [8, 7] describes a subroutine-threaded Forth for the 68000 CPU. He improves the code by inlining small opcode bodies and converting virtual branch opcodes to single native branches. Our baseline context threading technique is similar to subroutine threading with optimizations, described by Curley; in this paper we present extensions that enable dynamic profiling and more sophisticated optimizations based on that profiling.

Significant work has gone into examining the specific problem of path profiling including the classic work by Ball and Larus [4] which demonstrates an efficient intra-procedural path profiling technique using path enumeration. The problem of limiting instrumentation overhead is often handled with sampling. For example Traub et. al. suggest using self-modifying code to periodically insert and remove profiler hooks [20]. Our technique naturally enables this type of ephemeral profiling, as we describe in Section 4.3. Arnold et. al. suggest instead that code versioning be used to control the execution of profiled code [2, 1].

Whaley's partial method compilation technique [23] finds the *not-rare* basic blocks within hot methods. His dynamic optimizer identifies hot code in two phases: first discovering hot methods, then flagging the executed blocks in those methods. Counters are placed at both method entry points and loop back edges. When the method counter passes a threshold the entire method is compiled using a baseline compiler. Then, the baseline compiler resets each counter and instruments all the method's basic blocks so that they are flagged when executed. When the method counter passes a second threshold all code which was at some point executed is considered *not-rare* and compiled by the optimizing compiler. We describe our implementation of partial method selection in Section 4.4.3

HP Dynamo [3] is a system for trace-based runtime optimization of statically optimized binary code. Dynamo emits native code corresponding to commonly executed dynamic sequences of instructions detected using a speculative trace selection heuristic called SPECL. One would expect that this optimization would be highly applicable to threaded interpreters. Sullivan et al. [19] tested this idea by applying Dynamo to a Java virtual machine. Initially the resulting performance was poor because Dynamo could not distinguish between the different runtime contexts of the various bytecodes. They handled this problem by exposing the `vPC` to Dynamo, giving it enough information to generate traces across bodies. We present our implementation of SPECL trace selection in Section 4.4.1.

Software trace caches are efficient structures for dynamic optimization. In [6] Bruening and Duesterwald compare execution time coverage and code size for three dynamic optimization units: method bodies, loop bodies, and traces. They show that method bodies require significantly more code size to capture an equivalent amount of execution time than either traces or loop bodies. Further, loop bodies were able to capture more instruction stream coverage than traces. However, unlike our implementation, their trace detection mechanism did not inline method calls unless the entire method is compiled. In Section 5, we give a similar analysis comparing methods, traces and partial methods.

4 Design and Implementation

In this section, we describe how we modified two direct threaded virtual machines, SableVM 1.1.8 and OCaml 3.08, on two platforms (P4 and PPC). We describe the changes that were necessary to the interpreters themselves and how we loaded, profiled, and detected regions, as well as how we generated code.

The guiding principle behind our design is to maintain the ability to generate specialized code for flexible program regions, and to switch between this specialized code and baseline interpretation at designated program points, not just at method boundaries. As a secondary design principle, we sought to defer work whenever possible. In some cases, we can avoid the work altogether, saving both time and space overheads. For example, there is no need to instrument instructions that are never executed. In other cases, performing work lazily allows us to make use of dynamic information that was not available earlier, as with the generation of code for forward branches in context threading (described in Section 2.3.1).

4.1 Key Infrastructure

There are three key pieces of our infrastructure that together enable the flexible selection of program regions and mixed mode execution: pure subroutine threading, soft linking, and the insertion of generated code using *interposers*.

First, all bytecode bodies are implemented with *pure subroutine threading*, which means that all bodies end with a return instruction including those that implement virtual branch instructions. As a consequence, these branching bodies only update the virtual PC (`vPC`) but must depend on generated code following the return to actually perform the control transfer. Modifying the bytecode bodies in our implementations was straightforward, partially because the VMs we used already ended bodies with convenient C preprocessor macros. Thus, in many cases we only needed to replace the macro definition. Note that we do *not* end each body with a C language `return` statement. Instead we make use of `gcc` language extensions to embed native assembly instructions (the `asm volatile` statement). On Intel hardware, each body executes a native `ret` instruction, while a native `blr` instruction is used on the Power PC.

The second critical piece of our infrastructure is what we call *soft linking*. This means simply that the current implementation for any virtual instruction or program region can be dispatched by branching to the address stored in the corresponding slot of the DTT. In our system, as in direct threading, the `vPC` is a pointer into the DTT. Hence, when we refer to the DTT element corresponding to the current virtual instruction we more precisely mean the DTT element referred to by the current `vPC`. The first element in the DTT for each virtual instruction is an address, followed by any arguments needed by that instruction. In direct threading, the address is that of the bytecode body to dispatch, however, under CT it is the address of generated code that dispatches the virtual instruction. This is a subtle, but critical distinction. Because the DTT now points to generated code instead of referring directly to bodies, we can insert arbitrary code around the dispatch of every virtual instruction (which we call interposers). In our system, we have decreed that the DTT and the `vPC` must be valid on entry to generated native code regions. Thus, we can always dispatch a virtual program region, whether it turns out to be a single body or a large region, by branching to the address of the DTT element indexed by the current `vPC`. Initially, the code that brackets the pure subroutine threaded bodies ends with a soft link – an indirect branch through the DTT – to reach the dispatch code for the next virtual instruction. It is this mechanism that allows virtual branch instructions to end with a simple return after updating the `vPC`.

The final key piece of our infrastructure is the use of *interposers* around the "real work" of program regions (recall that a region may be as small as an individual bytecode body). The purpose of interposers is to provide convenient hooks to call arbitrary C functions at interesting program points. The choice of interesting points evolves during the execution of the program as regions of differing shapes are detected and optimized code is generated for them. Initially, we consider every virtual instruction "interesting", since we have no information about execution. The basic structure of a generic interposer is shown in Figure 2. Conceptually, all interposers consist of pre- and post-workers that bracket the real work of the region, followed by a soft link to the next region. In practice, some of these pieces may be omitted by particular flavours of interposer.

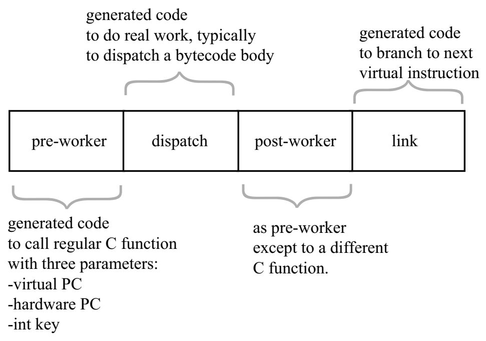

Figure 2: Interposers

The pre- and post-workers are regular C functions which the interposers always call with the same three parameter protocol. The first parameter is the `vPC`, or in other words, the DTT slot corresponding to the currently executing virtual instruction. The second parameter is the real hardware PC, or equivalently, the address of the start of the interposer. Together, these first two parameters define a point in the execution of the virtual program, and allow the C function to inspect and modify either the DTT or the interposer itself. We call the third parameter the instrumentation worker's *payload*. A payload is an arbitrary `int` value that is generated as part of the interposer. Example uses of this key value are to identify the virtual opcode, or to provide the address of a profiling data structure to be used by the worker function (as we describe in Section 4.2).

In most cases, the real work is one or a series of bytecode bodies; in the simplest case, it is a single native call instruction to dispatch one of the pure subroutine threaded bodies. In the following sections, we show how interposers are used to bootstrap the execution of a virtual program, and as part of our region selection mechanism.

4.2 Initialization and Dispatch

Having changed the virtual machines to support subroutine threading, soft linking and interposers, it was necessary to also modify the process of loading a program or program method into the virtual machine.

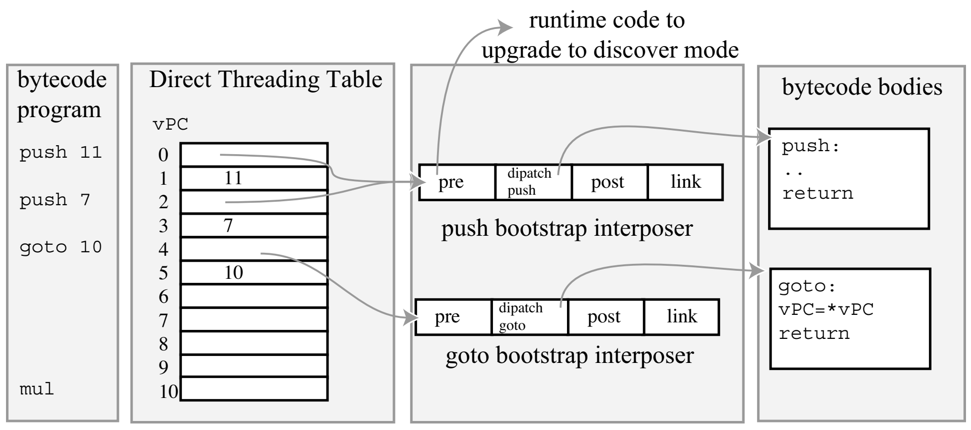

Figure 3: System initialization using bootstrap interposers

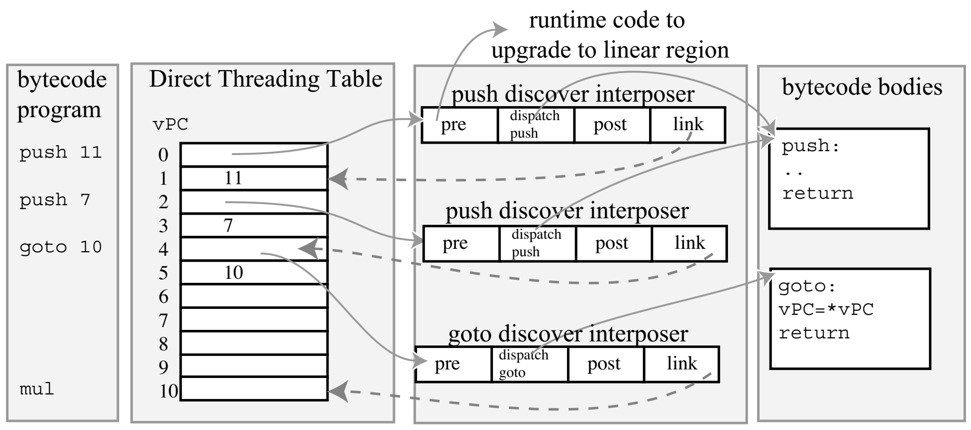

Figure 4: System initialization showing discovery interposers

Conceptually, this process of loading a method changes considerably. A direct threaded interpreter builds a direct threading table, or DTT, for each virtual method before it is run. For each virtual instruction, this process maps the virtual opcode to the address of the corresponding bytecode body (entering this address in the DTT), and then copies the immediate parameters for that instruction into the DTT. When loading is complete the direct threaded interpreter executes a computed goto to the first bytecode in a method and thereafter dispatch is handled by the bodies themselves. In contrast, a subroutine threaded bytecode body is dispatched by a direct call instruction – something that can only be conveniently done from generated code.

Our runtime has support for dynamic code generation so we can bootstrap more incrementally. This is advantageous because we can quickly generate very simple code for each instruction as it is loaded, while leaving hooks to add profiling or generate optimized code for only those instructions that are actually executed. Instead of generating a native call for every virtual instruction as a method is loaded, as we did in CT, we generate a small set of bootstrap interposers, one for each virtual opcode. The DTT entry for each virtual instruction is set to point to the interposer corresponding to its opcode. Figure 3 shows a fragment of a method that has just been loaded, illustrating the relationship between the DTT, bootstrap interposers, and bytecode bodies. Note that the two instances of the `push` instruction both link to same bootstrap interposer. Because these interposers are per-opcode, not per-instruction, there are many fewer of them to generate, saving both time and space during loading. The pre-worker for the bootstrap interposer is a call to a function that can perform additional work for those instructions that are executed; the integer value passed to the worker in this case is the opcode of the virtual instruction.

In our implementation, when the bootstrap interposer for each virtual instruction is executed for the first time, the pre-worker causes a new stub of instrumented dispatch code to be generated. Space is allocated to hold per-instruction profiling data, and the address of this structure is generated into the new interposer to be passed to its pre-worker function. The virtual instruction's DTT entry is then overwritten with the address of the newly-generated interposer. Other (unexecuted) instructions with the same opcode continue to link to the original bootstrap stub. Figure 4 illustrates a scenario where the virtual instruction at `vPC` equal to 10 is about to be executed for the first time. The two `push` instructions and the `goto` have executed once but the `mul` has not. Consequently, these three virtual instructions are now implemented using discovery interposers (note that the two `push` instructions now each have a distinct interposer) but the `mul` is still implemented by a bootstrap interposer. We describe the use of the discover interposer in detail in the following section. Simple logging interposers to support debugging could also replace the bootstrap interposers.

4.3 Linear Regions

In the previous section, we described how interposers are used to bootstrap code and how they can dynamically be replaced. As a consequence, the state of a virtual instruction is partially captured by the specific interposer that implements it at any given instant in time. For instance, a bootstrap interposer only executes once, the first time each virtual instruction is dispatched. In this section we describe how we exploit this aspect of our system to identify linear regions. A linear region is a linear sequence of virtual instructions ending in a branch. They have only one entry point, so in some cases they will be a concatenation of multiple basic blocks from the flow graph of the virtual program. In these cases, some virtual instructions may appear in multiple linear regions. For example, when execution falls through from one basic block to its successor, our system may detect two linear regions: one containing the concatenation of the two blocks, and the other containing just the successor.

4.3.1 Detection

Linear regions are identified by discovery interposers. The situation whereby execution reaches a discovery interposer indicates one of two possible actions. First, if no linear region is currently being collected then one should be started with the instruction as its entry point. Otherwise, the current instruction should be added to the current linear region. When the discovery interposer for a branching virtual instruction is encountered, execution has reached the last virtual instruction in the current linear region. Code to implement the region is then generated and the DTT slot corresponding to the entry point of the region is rewritten to point to the new code. Consequently the discovery interposers for instructions in the region will never be dispatched again, apart from exceptional control flow.

We implement linear regions with a *linear block* which does real work and a *region* interposer that wraps a native call to the linear block with further instrumentation. The linear block is simply a sequence of direct call instructions – one to dispatch each of the bytecode bodies identified as part of the region. The linear block ends with a return. Hence, it is reasonable to think of a linear block as the body of a new, dynamically identified virtual instruction.

Figure 5 illustrates the situation after a linear block has been built and is ready to execute. The DTT slot corresponding to the entry point to the linear region, at `vPC` 0 in the figure, is set to point to the region interposer. This means that any code that attempts to dispatch the virtual instruction at the entry point of the newly installed region will dispatch the linear block instead. When all the virtual instructions in the linear block have executed control returns to the region interposer, which links to the destination indicated by the `vPC`. In the hypothetical scenario of Figure 5 the linear region was ended by a `goto 10` and so the destination of the final link will be whatever is pointed to by the tenth slot of the DTT.

4.3.2 Sharing

Our block sharing strategy aims to identify when a newly-detected linear region can reuse a linear block created earlier. Our approach is to build an association between sequences of virtual instructions and linear blocks using a hash table. To this end we define a simple hash function that hashes the virtual instructions in a linear region to an integer. Then, as each linear region is identified, we check the hash table to see if it has been seen before. If the current linear region is identical to one in the hash table, we need not generate a new linear block. Instead, we generate only a new region interposer wrapped around the (now) shared linear

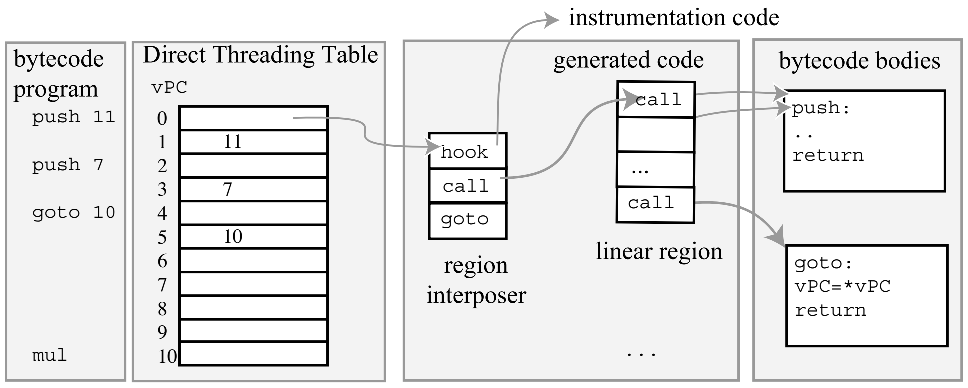

Figure 5: Implementation of Linear Region

block found in the hash table. Our decision to break the implementation of linear blocks into two interposers was motivated by this sharing. The strategy we adopt for shared linear blocks is similar to that used by Piumarta and Riccardi for performing selective inlining at load time [13]. In effect, our shared linear blocks are run-time selective inlining.

4.3.3 *Dynamic Linking*

Although soft linking is very powerful, it is also inefficient since indirect branches can strain the branch prediction resources of a modern pipelined CPU. Hence we would like to rewrite soft links as direct branches whenever possible. We will refer to direct branches used to link from one region of generated code to another as *hard links*. We have developed a mechanism, which we call *lazy linking* to convert soft links into hard links at run time.

Individual bytecode bodies depend on the interposers that dispatch them to soft link (using an indirect branch) to their successor. Similarly, regions of generated code depend on their region interposer to link to their successor. To this end, instead of a soft link, a region interposer ends with a direct branch to a lazy link interposer. The lazy link interposer examines the predecessor and successor of the edge and, if both are regions of generated code, rewrites the predecessor to end with a hard link. In fact, it is possible that the successor will be eventually be a generated region but the first time the lazy link is executed it has yet to be generated. In this case the lazy link interposer executes a soft link.

4.4 Selection of Larger Regions

Linear regions represent the simplest type of region that can be detected and optimized. In this section we show how larger regions with varying properties can be detected and generated using the same infrastructure.

4.4.1 *Traces*

We have implemented run time selection of traces using the SPECL heuristic described in [3]. Our implementation investigates the selection and dispatch of traces. We do not currently generate quality native code for them, we implement them using a CT-based approach. Whereas we implemented linear regions using linear blocks (sequences of calls to bytecode bodies), we implement traces as *trace blocks*, which are sequences of calls to linear blocks.

In our system trace selection is very similar to linear region discovery. Trace selection is initialized by instrumentation hooks called from the linear region interposer, as is illustrated in Figure 5. Instrumentation examines the vPC set by each linear region and, as called for by Dynamo's SPECL trace selection heuristic, if the branch is a reverse branch (i.e. a smaller vPC) it is counted. When the target becomes hot, trace generation begins. During trace generation, linear regions are appended to a list as they are visited. When a trace end condition is reached the region list is used to generate the trace. Figure 6 illustrates the generated trace's structure. The similarity between trace blocks (sequences of dispatches of linear blocks) and linear blocks (sequences of dispatches of bytecode bodies) is striking. This is not an accident, it is an elegant consequence of packaging linear blocks and bytecode bodies as callable regions of code.

Trace exits may occur between adjacent linear blocks. Though the blocks in a trace tend to fol-

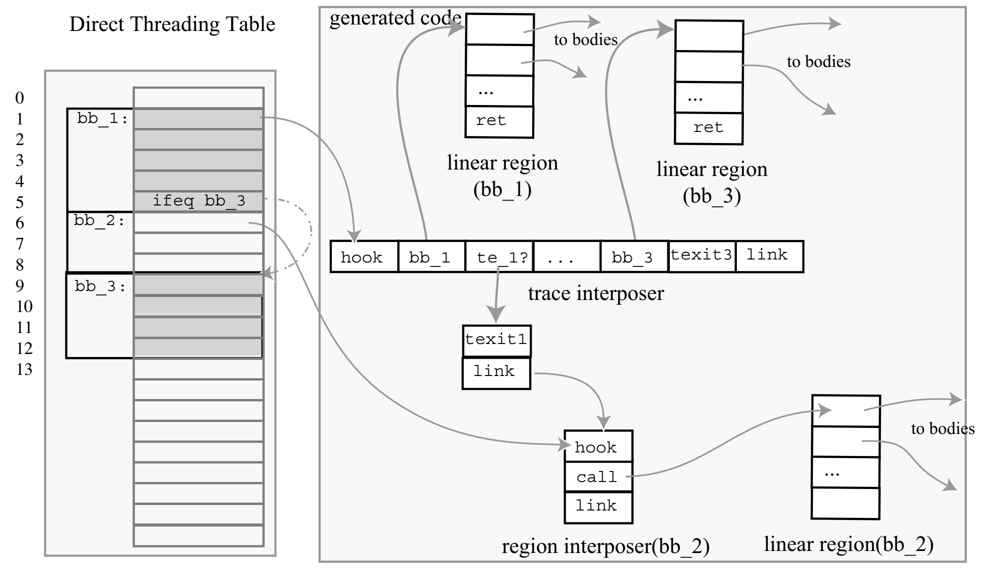

Figure 6: Implementation of a trace illustrating it is a sequence of linear blocks

low the particular execution path that occurred during trace generation, at run time we must generate guard code against execution diverging from that path. Hence, after running each linear block, generated code checks if the next block to be executed is in the trace. If not a trace exit must occur.

Recall that we have packaged all bytecode bodies as pure subroutines. Thus, the `vPC`, as set by the last (virtual branch) bytecode body in each linear block, determines its successor. Hence, in our system, we simply generate code into the trace block that checks that the `vPC` is the one we expect. Consider the trace exit `te_1` illustrated by Figure 6; the last virtual instruction executed by the first linear block (`bb_1`) ends with `ifeq`, a virtual conditional branch. After the block corresponding to `bb_1` returns, it has either set the `vPC` to `&DTT[9]`, the entry point of `bb_3`, or else it has diverged from the trace by setting the `vPC` to `&DTT[6]`. The presence of a valid `vPC` enables us to generate almost trivial native code for `te_1` – a compare immediate of the `vPC` to `&DTT[9]`. followed by a conditional branch to a specific *trace exit interposer*. The SPECL heuristic requires us to generate a new trace when a trace exit becomes hot. For this purpose, a trace exit executes a trace profiling hook and then lazy links to the off-trace code.

4.4.2 Hot Methods

Hot method selection is carried out by linear region interposer instrumentation. Figure 5 illustrates how a call to a block profiling hook is made *before* the dispatch of a linear block. Method selection is done from a similar profiling hook interposed *after* the execution of each linear block. There, we detect and count backwards branches (to detect methods that may not be frequently called yet contain hot loops) and entry points of new methods. Once a method has been detected as hot, a potentially large interposer is created with a direct call of the bytecode body corresponding to each virtual instruction in the method. This is the technique we describe as subroutine threading in [5].

4.4.3 Partial Methods

Partial method selection is a heuristic described in [23] designed to generate code for a method while leaving out cold code. Partial methods are created using a two stage process. First, the above method selection heuristic is used to find hot methods. Second, hot methods are instrumented such that every basic block executed during some training period (2000 invocations in our implementation) is recorded. Third, the method is recompiled including only blocks visited during the training period.

The first two stages of Whaley's technique can be implemented entirely in our instrumentation system. First, method selection is used unmodified to detect interesting methods. Second, while a method is training, we record the blocks it touches from the regular profiling hooks called by region interposers. We side-step many compilation-related

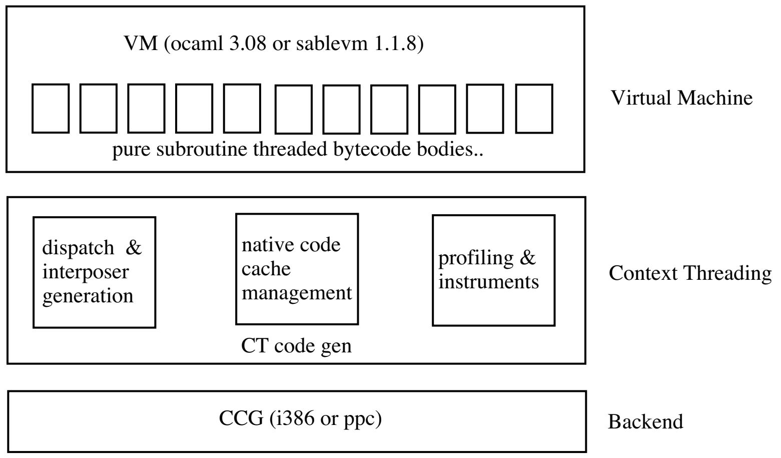
VM (ocaml 3.08 or sablevm 1.1.8)
pure subroutine threaded bytecode bodies..
Virtual Machine
dispatch & interposer generation
native code cache management
profiling & instruments
CT code gen
Context Threading
CCG (i386 or ppc)
Backend

Figure 7: Layered architectures of CT VM

challenges of the third stage above and use context threading to build these methods. The resulting partial method interposer is almost identical to a trace interposer. However, after dispatching each linear block in the partial method we check the `vPC` to ensure that execution remains in the partial method. If so, then a direct branch reaches the destination within the partial method interposer. If not, then control branches to a method exit interposer. Method exit interposers are similar to trace exits.

Having described the key components of our infrastructure and how they support the detection and construction of different types regions, we now briefly discuss some implementation details.

4.5 Software Organization

We organized our implementation into several layers as illustrated by Figure 7. The bulk of our software exists below the virtual machine and supports runtime code generation, profiling, and code management. The CT code generation layer defines a machine independent interface that must be implemented for each target CPU. We layered our code generation backend, principally the generation of interposers, on top of Piumarta's CCG [**?**], a runtime assembler. (CCG is a C preprocessor that converts GNU syntax assembler into native code for PPC and P4. It was originally designed to generate the bytecode bodies for Squeak [18], a portable smalltalk [**?**] virtual machine.)

For example, the `genInterposer` function is declared with the same prototype on P4 and PPC but is implemented in CCG for each platform separately. Currently, we use CT to generate code for regions (for instance linear blocks in Section 4.3.1) of the virtual program. This generates a direct call instruction to dispatch each bytecode body in a region. Although this performs better than direct threading, an optimizing code generator would improve performance greatly. In the next phase of our research, we plan to augment our infrastructure with a optimizer and the CT code generation layer with a general purpose code generator.

5 Preliminary Experimental Results

The current implementation of our system includes only a simple code generator which does not improve the performance of generated code beyond that made possible by subroutine threading. For this reason we report the raw performance of our system only to illustrate that the overhead imposed by our region selection system is relatively small. Our system includes powerful profiling tools which we used to study the relationship between region shape and proportion of time spent executing regions.

Despite the great deal of profiling, code generation, and rewriting that occurs during region detection, on a Pentium 4 our system runs at about the same speed as the unmodified direct threading virtual machine. Unmodified, direct threading SableVM requires 843 elapsed seconds to run all ten of our java benchmarks. Our linear region selecting version requires 771 seconds. On the very much smaller OCaml benchmarks the situation is reversed. Unmodified OCaml requires 4.04 seconds whereas we run for 4.57 seconds. We draw two conclusions from these results. First, the overhead of our techniques are significant but not problematic. Hence on the tiny OCaml microbenchmarks the overhead of profiling and detecting linear regions is not possible to recoup. On the much larger Java benchmarks the overheads are less than the speedup that even our very simple subroutine threading code generator enables. Second, these data suggest that with an optimizing code generator our approach may be capable of very interesting performance.

One goal of this phase of our research is to form an impression of how the various shapes of compilation unit compare. Development organizations invest in mixed-mode execution environments because they believe that a region of virtual program will run much faster when translated to native code than it can be interpreted. Clearly we would like to interpret as little as possible while compiling as little code as possible. In [6] the "90-10" rule is described. The suggestion is that a mixed-mode system should strive to find the 10% of the virtual code that leads to executing generated code 90% of the time. We believe this principle is sound, though the precise levels are a bit dated – we test the "95-5" rule and the "99-1" rule also.

We wish to investigate how the proportion of virtual instructions executed from regions of varying shapes relative to virtual instructions executed overall varies as more virtual code is translated into native code. Our metric of how much virtual code has been compiled is the ratio of virtual instructions translated to the number of virtual instructions loaded in all active methods.

Each stacked bar in Figure 8 represents the characteristics of a region shape on a single benchmark. The stack shows how much code needs to be compiled in order to achieve a given ratio. The height of the bottom stack represents the "90-10" rule. Its height indicates how much code needs to be translated to cause 90% of the execution to come from the region. For example, the height of the bottom stack (in solid black) of the rightmost bar in the figure indicates that slightly more than 5% of the code of javac needs to be compiled to attain 90% of the execution from the traces. Note also that the absence of a white stack (evident on top of most other stacks) indicates that traces never attained 99% execution of javac.

One point that must be raised is that these results do not inform *which* 90% of javac should be converted to traces. Figure 8 effectively assumes that an oracle is available to choose the most frequently executed traces. Naturally these were calculated after the fact, but in real life we face the additional challenge of designing a heuristic to select code. Hence, when a real heuristic is used to select traces we should expect more code to be compiled.

On the X-axis we have sorted stacks in two ways. The coarsest level of sorting is by shape. Each cluster of stacks represents the results for a shape. Within each cluster we have ordered benchmarks, the standard SPEC98 suite [17], scimark [14], and soot[21], a bytecode to bytecode optimizing compiler, such that those we expected to be strongly looping (compress, db, etc) are on the left whereas javac, etc, are on the right. This means that within each cluster stacks tend to have an upward trend towards the right as unpredictable benchmarks require more code to cover.

We will discuss the various shapes moving from left to right across the figure. On the left, we have pure linear block generation and cached linear block generation (our runtime version of selective inlining). Comparing these two heuristics we see the advantage offered by avoiding the regeneration of identical blocks. For the strongly looping benchmarks on the left of each cluster, execution is dominated by the, mostly unique, blocks of the inner loops. The interesting exception is mpegaudio, which, due to obfuscation, has many unique and relatively rarely executed instruction sequences. However, the impact becomes much more noticeable for the less predictable benchmarks. At 99%, two thirds of the hottest blocks are shared, and are eliminated by caching.

The next three clusters show execution density for the larger execution regions. The central bar cluster shows delayed whole method compilation, built with an execution delay of 2000 method entries or back-edge executions. The next cluster presents Whaley's partial method selection, using the suggested initialization and detection delays of 2000 and 25000 of the above executions, respectively. As methods are correctly built after the initial delay, the time spent executing these methods is included. On the right, we present the distribution for SPECL traces built with a recording delay of 50 entry-point executions.

Comparing the shapes we see that, with the interesting exception of the obfuscated mpegaudio, they all fare well for strongly looping benchmarks. Only tracing with its low initial threshold is able to capture the full 99% of the execution of mpegaudio. This is due to the longer training period used for

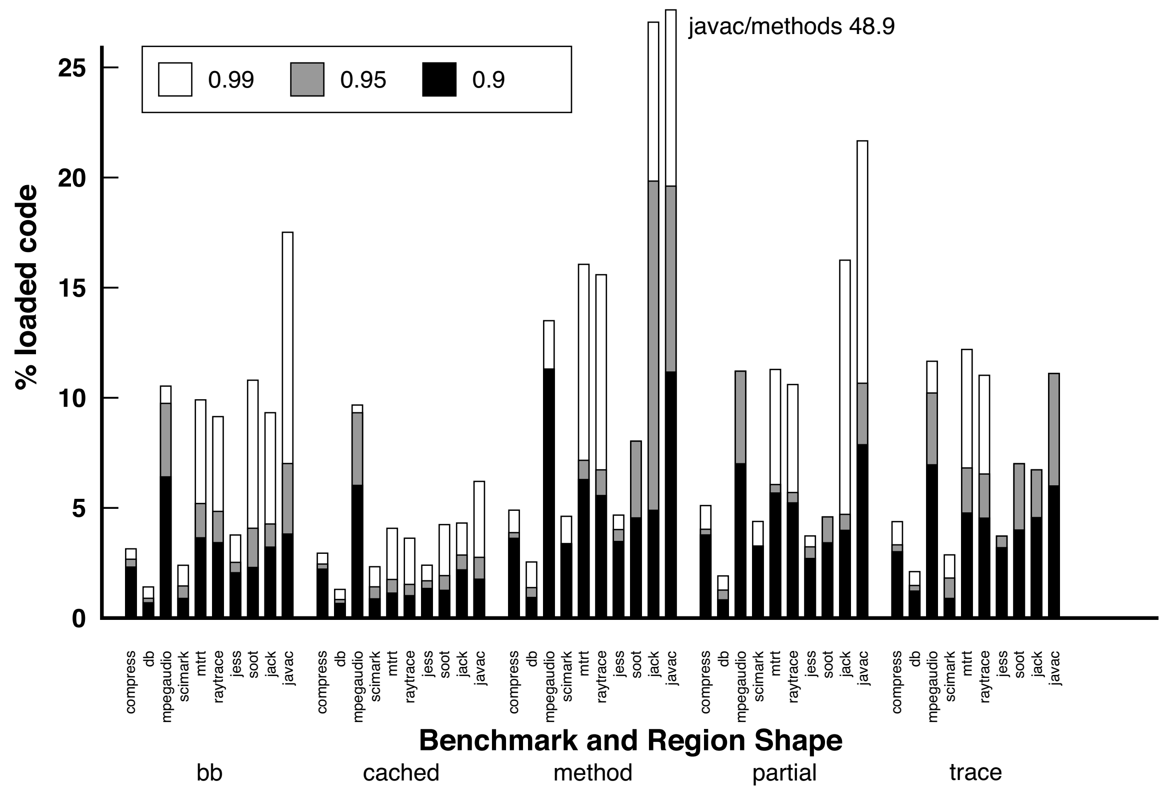

Figure 8: SPECjvm Java benchmark results comparing various shapes of region. Each cluster represents a shape of region to compile. Each stacked bar represents a specific benchmark. The height of each stack represents the minimum proportion of all virtual instructions that must be compiled in order for a given proportion of all virtual instructions executed to be from the compiled code cache.

methods and partial methods combined with mpegaudio's multiplicity of methods. Another interesting benchmark is soot, for which none of the shapes is able to capture the full 99%. One reason for this might be that we execute soot on a single small class file, limiting amortization opportunities. For the other benchmarks, the advantage of partial methods over methods is clear, with significant decreases in the amount of code generated. Examining traces we see that for many of the more predictable benchmarks they outperform both methods and partial methods. However, for the least predictable benchmarks tracing never reaches the full 99% though examining the detailed data we found traces do in total reach around 98%.

In summary, although modern JIT compilers restrict themselves to methods it appears that non-static shapes may be able to compile significantly less code while executing a similar proportion of the time from their native code cache.

6 Conclusions

Mixed-mode execution is interesting because it provides a way of incrementally raising the performance of virtual machines. High performance, Just in Time (JIT) compilation techniques make use of mixed-mode execution that is faster than either interpretation or compilation alone. However, creating the infrastructure for a JIT requires a large upfront investment in development effort. The challenge of method based JIT compilation is that by using a language defined static unit of compilation you must be able to handle all the features of that language. Thus, before you can optimize *anything* you have to be able to compile *everything*. Ideally, the architecture for a mixed-mode VM could detect and execute a variety of shapes of hot regions of a virtual program.

Our vision of the lifecycle of an interpreted language is that we would initially deploy a context threaded interpreter. Then, when performance issues arise, we would deploy a modest mixed-mode system which would select a few small, specific regions, and generate good native code for those. As the need arose, we would incrementally increase the size and generality of the regions we could compile.

In this paper, we have described how our architecture allows for a variety of mixed mode execution shapes. We believe our system makes it relatively simple to support a wide variety of compilation unit shape because of a beneficial interaction of subroutine threading, virtual PC and direct threading table maintenance.

Subroutine threading allows us to intersperse the dispatch of bytecode bodies with generated native code. This means that we never have to generate code for functionality that is already provided by bytecode bodies. Nor is it precluded, the cost of the duplication of effort need not be paid unless the benefits of the performance gains justify it. Subroutine threading naturally provides good interposition opportunities. All of the profiling techniques we have described depend on interposing calls to runtime routines around bytecode dispatch. It seems to be particularly important that *pure* subroutine dispatch is used, where virtual branch instructions are pure calls also, modifying the virtual PC and returning to their caller. This makes it very simple to interpose on the control flow edges of the virtual program. Our experience so far indicates that this simplifies region selection and generation.

Retaining the direct threading table and maintaining the `vPC` at region boundaries enables our lazy linking technique. The DTT was originally devised to manage virtual control flow between bytecode bodies in a direct threaded interpreter. In exactly the same way, it allows us to manage virtual control flow between regions of code. We initially connect regions with easy to generate lazy links and rewrite them as hard links later when the destination is known. This allows us to avoid the complexity of handling forward references and allows us to generate lazy links to destinations that may or may not have code generated for them in the future.

The performance data we report indicates that the overhead of our region selection and code generation is modest. In fact, several significant Java benchmarks run faster than the original direct threaded implementations even though profiling is on. A very interesting question is the extent to which the constraints of virtual PC maintenance and bytecode body reuse will impede the optimization of regions. Hence, in the very near future we will turn our attention on how to generate high performance code.

7 Acknowledgements

The ideas presented in this paper have benefited from discussions with Kevin Stoodley, Mark Stoodley, and Benjamin Vitale. The research project is supported by grants from IBM CAS, CITO and NSERC.

8 About the Authors

Mathew Zaleski is currently a PhD candidate in the department of computer science at the University of Toronto. In the preceding 15 years Mathew filled a number of technical, management and business positions in the software industry including at Immersant, Alias Research and IBM.

Marc Berndl is currently working for Google, on leave from the PhD program in the Department of Computer Science at the University of Toronto. He received an MSc from McGill University in 2004.

Angela Demke Brown is an Assistant Professor in the Department of Computer Science at the University of Toronto. She received her MSc from the University of Toronto and her PhD from Carnegie Mellon University. Her research interests include run-time optimization and operating systems.

9 References

[1] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of Java. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 111–129, New York, NY, USA, 2002. ACM Press.

[2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 168–179, New York, NY, USA, 2001. ACM Press.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2000. ACM Press.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[5] M. Berndl, B. Vitale, M. Zaleski, and A. Demke Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *CGO '05: Proceedings of the Third International Symposium on Code Generation and Optimization*, pages 15–26, Washington, DC, USA, 2005. IEEE Computer Society.

[6] D. Bruening and E. Duesterwald. Exploring optimal compilation unit shapes for an embedded just-in-time compiler. In *FDDO-3: Proceedings of the Third ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2000.

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

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

[9] M. A. Ertl. *Implementation of Stack-Based Languages on Register Machines*. PhD thesis, Technical University of Vienna, Austria, 1996.

[10] M. A. Ertl and D. Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 403–412, London, UK, 2001. Springer-Verlag.

[11] M. A. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 278–288, New York, NY, USA, 2003. ACM Press.

[12] 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, March 2002.

[13] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 291–300, New York, NY, USA, 1998. ACM Press.

[14] R. Pozo and B. Miller. *SciMark: a numerical benchmark for Java and C/C++*, 1998. http://www.math.nist.gov/SciMark.

[15] T. A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 322–332, New York, NY, USA, 1995. ACM Press.

[16] E. D. Rather, D. R. Colburn, and C. H. Moore. The evolution of Forth. *ACM SIGPLAN Notices*, 28(3), March 1993.

[17] SPEC JVM98 benchmarks, 1998. http://www.spec.org/osg/jvm98/.

[18] Squeak project information. http://www.sourceforge.net/projects/squeak.

[19] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic native optimization of interpreters. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 50–57, New York, NY, USA, 2003. ACM Press.

[20] O. Traub, S. Schecter, and M. D. Smith. Ephemeral instrumentation for lightweight program profiling. Technical report, Harvard, June 2000.

[21] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[22] B. Vitale and T. S. Abdelrahman. Catenation and specialization for tcl virtual machine performance. In *IVME '04: Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators*, pages 42–50, New York, NY, USA, 2004. ACM Press.

[23] J. Whaley. Partial method compilation using dynamic profile information. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 166–179, New York, NY, USA, 2001. ACM Press.