YETI: a graduallY Extensible Trace Interpreter

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

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

Kevin Stoodley
IBM Toronto Software Lab
stoodley@ca.ibm.com

Abstract

The design of new programming languages benefits from interpretation, which can provide a simple initial implementation, flexibility to explore new language features, and portability to many platforms. The only downside is speed of execution, as there remains a large performance gap between even efficient interpreters and mixed-mode systems that include a just-in-time compiler (or *JIT* for short). Augmenting an interpreter with a JIT, however, is not a small task. Today, JITs used for Java™ are loosely-coupled with the interpreter, with callsites of methods being the only transition point between interpreted and native code. To compile whole methods, the JIT must duplicate a sizable amount of functionality already provided by the interpreter, leading to a "big bang" development effort before the JIT can be deployed. Instead, adding a JIT to an interpreter would be easier if it were possible to leverage the existing functionality.

In earlier work we showed that packaging virtual instructions as lightweight callable routines is an efficient way to build an interpreter. In this paper we describe how callable bodies help our interpreter to efficiently identify and run traces. Our closely coupled dynamic compiler can fall back on the interpreter in various ways, permitting an incremental approach in which additional performance gains can be realized as it is extended in two dimensions: (i) generating code for more types of virtual instructions, and (ii) identifying larger compilation units. Currently, Yeti identifies straight line regions of code and traces, and generates non-optimized code for roughly 50 Java integer and object bytecodes. Yeti runs roughly twice as fast as a direct-threaded interpreter on SPECjvm98 benchmarks.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—interpreters, incremental compilers

General Terms Design, Languages, Performance

Keywords interpreter, JIT compiler, mixed-mode, traces

1. Introduction

An interpreter is an attractive way to support an evolving computer language, making it easy to test and refine new language features. The portability of an interpreter also allows a new language to be widely deployed. Nevertheless, these interpreted language implementations generally run much more slowly than compiled code. To get the best of both worlds, today's high-performance Java implementations run in *mixed-mode*, that is, combining interpretation with dynamic just-in-time (JIT) compilation. Given the success of this strategy for Java, why are many useful languages like Python, JavaScript, TCL and Ruby not implemented by mixed-mode systems? We believe the problem is due to the structure of current mixed-mode virtual machines. We present an approach that more closely integrates interpretation and JIT compilation, allowing for the gradual evolution of a mixed-mode runtime system.

We focus on *virtual machine interpreters* in which source code is compiled to a *virtual program* or *bytecode* representation (i.e., a sequence of *virtual instructions* and their operands). Typically, virtual instructions are described as though provided by real hardware, but in fact the virtual machine implements each with a block of code, called the *virtual instruction body*, or simply *body*. The interpreter executes the virtual program by *dispatching* each body in sequence.

1.1 Challenges of Evolving to a Mixed-Mode System

Current JIT compilers (or *JITs* for short) are method-oriented, that is, the JIT must generate code for entire methods at a time. This leads to two problems. First, if the construction of the JIT is approached in isolation from an existing interpreter, the project becomes a "big bang" development effort where the code generation for dozens, if not hundreds, of virtual instructions is written and debugged at the same time. Second, compiling whole methods compiles cold code as well as hot. This complicates the generated code and its runtime.

The first issue can be dealt with by more closely integrating the JIT with the interpreter. If the interpreter provides a callable routine to implement each virtual instruction body, then, when the JIT encounters a virtual instruction it does not fully support, it can generate a call to the body instead [22]. Hence, rather than a big bang, development can proceed in a sequence of stages, where JIT support for one or a few virtual instructions is added in each stage. Although typical interpreters do not currently provide callable virtual instruction bodies we recently demonstrated that callable bodies can be dispatched very efficiently [2].

The second issue, compiling cold code (i.e., code that has never executed), has more implications than simply wasting compile time. Except at the very highest levels of optimization, where analyzing cold code may prove useful facts about hot regions, there is little point compiling code that never runs. Moreover, cold code increases the complexity of dynamic compilation. We give three examples. First, for late binding languages such as Java, cold code likely contains references to program values that are not yet resolved. If the cold code eventually does run, the generated code and the runtime that supports it must deal with the complexities of late binding [25]. Second, certain dynamic optimizations are not possible without profiling information. Foremost amongst these is the optimization of virtual function calls. Since there is no profiling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VEE'07, June 13–15, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-630-1/07/0006...$5.00

information for cold code the JIT may have to generate relatively slow, conservative code. Third, as execution proceeds, cold regions in compiled methods may become hot. The conservative assumptions made during the initial compilation may now be a drag on performance. The straightforward-sounding approach of recompiling these methods is complicated by problems such as what to do about threads that are still executing in the method or threads that must return to the method in the future.

These considerations suggest that the architecture of a gradually extensible mixed-mode virtual machine should have three important properties. First, virtual bodies should be callable routines. Second, the unit of compilation must be dynamically determined and of flexible shape, so as to capture hot regions while avoiding cold. Third, as new regions of hot code reveal themselves, a way is needed of gracefully compiling and linking them on to previously compiled hot code.

1.2 Overview of Our Solution

One of our goals is to design an infrastructure that supports dynamic compilation units of varying shape. Just as a virtual instruction body implements a virtual instruction, a *region body* implements a region of the virtual program. Possible region bodies include single instructions, straight-line regions of code, methods, partial methods, inlined method nests, and traces (i.e., frequently-executed paths through the virtual program). A key idea is to package every region body as callable, regardless of the size or shape of the region of the virtual program that it implements. The virtual machine can then execute the virtual program by dispatching each region body in sequence.

Region bodies corresponding to longer sequences of virtual instructions will run faster than those compiled from short ones because fewer dispatches are required. In addition, larger region bodies should offer more opportunities for optimization. However, larger region bodies are more complicated and so we expect them to require more development effort to detect and compile than short ones. This suggests that the performance of a mixed-mode VM can be gradually improved by incrementally increasing the scope of region bodies it identifies and compiles. Ultimately, the peak performance of the system should theoretically be at least as high as current method-based JITs since, with enough engineering effort, regions identical to the inlined method nests used by method-based JITs could be supported.

To test out these ideas we extended JamVM, a cleanly implemented and relatively high performance Java interpreter [16], to be a trace-oriented dynamic compilation system. We chose Java for our research, rather than a new language with no JIT, because we felt it essential to compare the performance of our prototype with existing high performance systems.

We built our prototype, Yeti, (graduallY Extensible Trace Interpreter) in five phases: First, we repackaged all virtual instruction bodies as callable. Our initial implementation executed only single virtual instructions. Second, we identified an approximation to basic blocks, straight line sequences of virtual instructions, or *linear blocks*. Third, we extended our system to identify and dispatch *traces*, or sequences of linear blocks. Traces are significantly more complex region bodies than linear blocks because they must accommodate virtual branch instructions. Fourth, we extended the trace system to link traces together. Up to this point all virtual instructions in traces are executed by dispatching the corresponding virtual instruction bodies. This system can be thought of as a trace-oriented interpreter. In the fifth and final stage, we implemented a naive, non-optimizing compiler to compile the traces. Our compiler currently generates PowerPC® code for about 50 virtual instructions, generating calls to virtual instruction bodies for the remainder. This system is a trace-oriented mixed-mode virtual machine.

We chose traces because they have several attractive properties: (i) they can extend across the invocation and return of methods, and thus have an inter-procedural view of the program, (ii) they contain only hot code, (iii) they are relatively simple to compile as they are *single-entry multiple-exit* regions of code, and (iv), as new hot paths reveal themselves it is straightforward to generate new traces and link them onto existing ones.

These properties make traces an ideal region body for an entry level mixed-mode system like Yeti is today. However, sophisticated region bodies assembled from linked traces may turn out to have all the advantages of inlined method nests but also side-step the overhead of generating code for cold regions within the methods. Finally, because of the way traces dynamically capture the active paths in a program our approach may (i) adapt more gracefully to phase changes in program behavior than traditional method-based systems [18], and (ii) take advantage of execution context, as is done in context sensitive inlining [13].

1.3 Outline of paper

We describe an architecture for a virtual machine interpreter that facilitates the gradual extension to a trace-based mixed-mode JIT compiler. We demonstrate the feasibility of this approach in a prototype, Yeti, and show that performance can be gradually improved as larger program regions are identified and compiled.

In Section 2, we present background and related work on interpreters and JITs, before describing the design and implementation of Yeti in Section 3. We evaluate the benefits of this approach in Section 4. Section 5 describes related work. Finally, we discuss avenues for future work and conclusions in Section 6.

2. Interpreter and JIT Background

To motivate the design choices we made in our system, we first review existing interpreter dispatch and JIT compilation strategies. We note that portability is an important property of interpreters, particularly for a new language implementation. Thus, it should be possible to build the source code base of the interpreter on a large number of platforms. On the other hand, dynamic compilers are intrinsically non-portable software, since they must generate platform-specific code. Some non-portable functionality may therefore be required by an interpreter to help it to integrate conveniently with the JIT. As we review various interpreter techniques, we comment on both their portability and suitability for gradual JIT development.

2.1 Interpreter Dispatch

The most straightforward way to implement an interpreter is with a dispatch loop that contains a `switch` statement with a `case` to implement the body of each virtual instruction. This technique is often used (e.g. in the JavaScript and python interpreters) because it uses only ANSI C features and so is very portable. However, it is slow because of the overhead of the dispatch loop and the switch. To address this overhead, more efficient dispatch mechanisms have been developed.

2.1.1 Direct Threading

As shown on the left of Figure 1, a virtual program is loaded into a direct-threaded interpreter by constructing a *list of addresses*, one for each virtual instruction in the program, pointing to the entry of the body for that instruction. We refer to this list as the *Direct Threading Table*, or DTT, and refer to locations in the DTT as *slots*. Virtual instruction operands are also stored in the DTT, immediately after the address of the corresponding body. The interpreter maintains a *virtual program counter*, or `vPC`, which points to a slot in the DTT, to identify the next virtual instruction to be executed and to allow bodies to locate their operands.

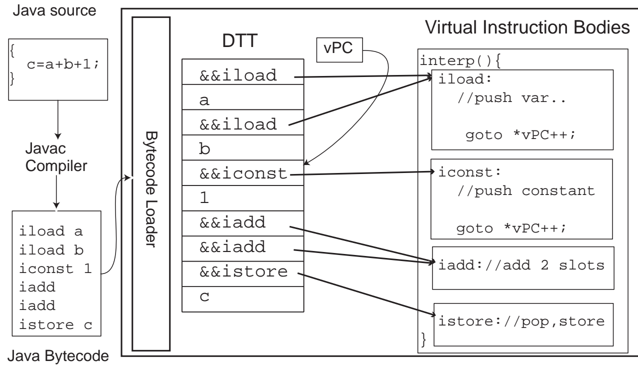

Figure 1. Direct Threaded Interpreter showing how Java source code compiled to Java bytecode is loaded into the Direct Threading Table (DTT). The virtual instruction bodies are written in a single C function, each with a separate label. The double-ampersand (`&&`) shown in the DTT is gcc syntax for the address of a label.

Interpretation begins by initializing the `vPC` to the first slot in the DTT, and then jumping to the address stored there. Each body then ends by transferring control directly to the next instruction, shown in Figure 1 as `goto *vPC++`. Although this requires fewer instructions than switch dispatch, the indirect branch generated from the `goto` is nonetheless expensive on modern CPUs because of branch mispredictions [10].

In C, bodies are identified by a `label`. Common C language extensions[1] permit the address of this label to be taken, which is used when initializing the DTT. The computed goto used to transfer control between instructions is also a common extension, making direct threading quite portable. Bodies packaged this way are not conveniently callable from generated code since the dispatch embedded in each body branches out of the control of calling code.

2.1.2 Direct Call Threading

It is also possible to write each body as a callable routine. A virtual method could be loaded as for direct threading, and a simple dispatch loop could then call each body by treating the `vPC` as a function pointer. Ertl [9] named this technique *direct call threading* because it is derived from direct threading.

We can create lightweight subroutines from direct threaded bodies by replacing the "`goto vPC++`" with code to increment the `vPC` and an inlined assembly return statement. This avoids the overhead of setting up and returning from a compiler-generated function call, but is somewhat less portable since a platform-specific return instruction must be used. We expect that this lightweight alternative has performance comparable to switch-based dispatch, since the destination of the indirect call is unpredictable, and we still have the dispatch loop overhead. Nonetheless, direct call threading is an interesting starting point. The dispatch loop provides a convenient way to add instrumentation, enabling the discovery of hot regions. Further, callable region bodies of arbitrary shape can be dispatched by simply rewriting slots in the DTT corresponding to their entry points.

2.1.3 Subroutine Threading

Subroutine threading, shown in Figure 2, is another modification of direct threading in which every virtual instruction body is called. It solves the branch prediction problems plaguing direct threading by

[1] GNU's `gcc`, as well as the C compilers produced by Intel, IBM and Sun Microsystems all support the label as address and computed goto extensions.

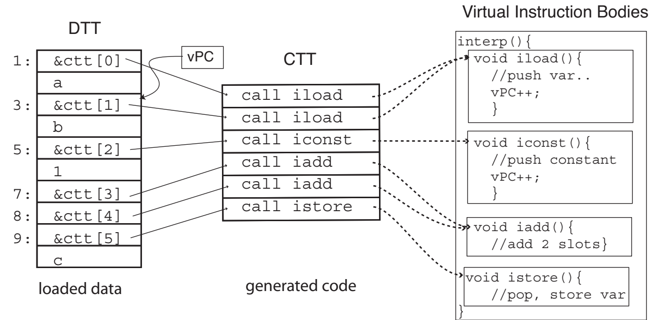

Figure 2. Subroutine Threaded Interpreter showing how the CTT contains one generated direct branch instruction for each virtual instruction and how the DTT points to generated code.

generating (at load time) a distinct *direct*, *relative* call to the body for each virtual instruction in the loaded program. We call the region of generated call instructions the *context threading table*, or CTT [2]. Since each call has a single destination, known when the call is generated, no prediction is needed. Return branch predictors, commonly available on modern microprocessors, remove mispredictions that are due to the return instructions. The direct threading table (DTT) is modified such that the first DTT slot for each virtual instruction points to the CTT slot holding the corresponding `call` instruction. This provides a level of indirection mapping the `vPC` to the *address of the generated code* that dispatches the current virtual instruction. Indirect virtual branch instructions (e.g., method invocation and return) need this level of indirection: they use the `vPC` to execute an indirect branch to the slot of the CTT corresponding to their destination.

To start executing any region of virtual code a subroutine threading interpreter jumps to the generated code at the slot of the CTT corresponding to the entry point of the region. Then, execution ping-pongs back and forth between the code in the CTT and the virtual instruction bodies. Reduced branch mispredictions lead to better performance than direct threading [2], but the machine code in the CTT is, of course, not portable. However, since only direct relative call instructions are generated, the machine dependency is simple and highly localized. In a sense, subroutine threading could be regarded as an extremely simple JIT. Generating a sequence of call instructions to implement a region of the virtual program is an appealing way to construct arbitrarily shaped region bodies, since the dispatch of individual instructions within the region is very efficient. However, doing so at load time may be wasteful, particularly for large methods containing many instructions that are never executed. We prefer to generate CTT code only for parts of the program that are known to execute.

2.1.4 Selective Inlining

Arguably, the best way to optimize virtual instruction dispatch is to eliminate it. Piumarta and Riccardi describe *selective inlining* [20] in which virtual instructions from the same basic block can be combined into a *superinstruction* at load time. The result is reduced dispatch overhead at run time in exchange for slightly higher load time overhead. Ertl and Gregg later showed that superinstructions reduce branch mispredictions [10]. Languages, like Java, that require runtime binding complicate the implementation of selective inlining significantly, but a variation of the technique for Java is deployed by SableVM [11]. The benefits of selective inlining suggest that basic blocks are a useful first step in identifying and generating code for region bodies larger than single instructions.

2.2 Traces

HP Dynamo [1, 8] is a system for trace-based runtime optimization of statically optimized binary code. Dynamo initially interprets a binary executable program, detecting interprocedural paths, or *traces*, through the program as it runs. These traces are then optimized and loaded into a *trace cache*. Subsequently, when the interpreter encounters a program location for which a trace exists, it is dispatched from the trace cache. If execution diverges from the path taken when the trace was generated then a *trace exit* occurs, execution leaves the trace cache and interpretation resumes. If the program follows the same path repeatedly, it will be faster to execute code generated for the trace rather than the original code. Dynamo successfully reduced the execution time of many important benchmarks. Significant binary optimization systems, including DynamoRIO [4], Mojo [5], Transmeta's CMS [6], and others, have since used traces.

Dynamo uses a simple heuristic, called Next Executing Tail (NET), to identify traces. NET starts generating a trace from the destination of a hot reverse branch, since this location is likely to be the head of a loop, and hence a hot region of the program is likely to follow. If a given trace exit becomes hot, a new trace is generated starting from its destination. Recently, Hiniker et al. [14] described improvements to NET that reduce replication and handle loops better.

Software trace caches are efficient structures for dynamic optimization. Bruening and Duesterwald [3] 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. This result, together with the properties outlined in Section 1.2, suggest that traces are a desirable region body for a gradually-extensible interpreter.

2.3 JIT Compilation

Modern JIT compilers can achieve much higher performance than efficient interpreters because they generate optimized code for potentially large regions of the virtual program. Typically these JITs and the interpreters with which they coexist are not tightly coupled [23, 17]. Rather, a profiling mechanism detects hot methods which are then compiled to native code. When the interpreter next attempts to invoke a method that has been compiled, the native code is dispatched instead. Although JIT compilation of entire methods has been proven in practice, it nevertheless has a few limitations. First, some of the code in a compiled method may be cold and will never be executed. Compiling this code can have only indirect benefits, such as proving facts about the portions of the method that *are* hot. Second, some of the code in a method may not yet have executed when the method is first compiled, even though it will become hot later. In this case the JIT has no profiling data to work with when it compiles the cold code. Handling these issues contributes to the large development effort that goes into creating a JIT.

JITs perform many of the same optimizations performed by static compilers, including method inlining and data flow analysis, both of which can be hindered by methods that contain large amounts of cold code, as observed by Suganuma et al. [24]. To deal with the problem, they modify a method-based JIT to allow selected regions within a method to be inlined, and rely on *on-stack replacement* [15] and recompilation to recover if a non-inlined part of a method is executed. Although avoiding cold code reduced compilation overhead significantly, only modest overall performance gains were realized. Whaley also describes a prototype that compiles partial methods [27].

An obvious way to investigate trace-based JIT compilation would be to retrofit traces into an existing method-based JIT, similar to the region-based approach. We looked into ways to do this and found our efforts complicated by many perfectly reasonable assumptions made by the developers of the JIT. For instance, the interpreter component of a method-based JIT expects to dispatch and return from generated code only at method invocation points. Similarly, the JIT assumes that all generated code is entered at the head of a method, executes to the end of the invoked method, and then abandons its stack frame. Traces, on the other hand, may begin and end at any virtual branch. On-stack replacement could probably be leveraged to handle trace exits, but much effort would be required because of the loose integration between the JIT and the interpreter. The upshot is that converting the code generator and optimizer of a method-based system to support traces is difficult, at best.

A JIT can also perform optimizations that require information obtained from a running program. A classic example of such a dynamic optimization addresses the cost of virtual method invocation, which is expensive at least in part because the destination depends on the class of the invoked-upon object. Polymorphic method invocation has been heavily studied and it is well known that most polymorphic callsites are *effectively monomorphic*, which means that at run time the invoked-upon object always turns out to have the same type, and hence the same callee is invoked all or most of the time [7]. Self [26] pioneered the dynamic optimization of virtual dispatch, an optimization that has great impact on the performance of Java programs today. With profile information, a JIT can transform a virtual method dispatch to a relatively cheap check of the class of the invoked-upon object followed by the inlined code of the callee. If the callsite continues to be monomorphic the check succeeds and the inlined code executes. If, on the other hand, the check fails, a relatively slow virtual dispatch must take place. Hölzle [15] describes how a polymorphic inline cache (PIC) can deal with an effectively polymorphic callsite that has a few hot destinations. More recently, aggressive speculative schemes have been built that assume callsites are monomorphic and then react, correcting their mistake, when the speculation is exposed as false [18].

A trace-oriented system should be able to apply the same optimizations. It can also afford to be more speculative for two reasons. First, fallback to interpretation is designed to be possible at any branch. Second, if speculation fails frequently enough, a new trace is generated from that point, needing less recompilation than an entire method.

3. Design and Implementation

An important goal of our design is to enable the gradual enhancement of an interpreter by incrementally increasing the size and scope of region bodies. We develop this design through a prototype called Yeti, based on the JamVM Java virtual machine. The relationship between the `vPC`, the DTT and generated code that exists in subroutine threading is also at the core of Yeti's representation of a loaded program.

Yeti runs a program by initially dispatching single virtual instruction bodies from an instrumented dispatch loop reminiscent of direct call threading. Instrumentation added to the dispatch loop detects region bodies, initially linear blocks, then traces, then linked traces. As region bodies are generated their addresses are installed into the DTT. Consequently the system speeds up as more time is spent in region bodies and less time on dispatch.

3.1 Instrumentation

In Yeti, as in subroutine threading, the `vPC` points into the DTT where each virtual instruction is represented as one or more contiguous slots. In Yeti, however, we add an extra level of indirection.

The first DTT slot of each instruction now points to an instance of a *dispatcher* structure. The dispatcher structure contains four key fields. The region body to be dispatched (initially the virtual instruction body, hence the name) is stored in the *body* field. The *preworker* and *postworker* fields store the addresses of the instrumentation routines to be called before and after the dispatch of the region body. Finally, the dispatcher has a *payload* field, which is a chunk of profiling or other data that the instrumentation needs to associate with a given vPC.

Although slow, a dispatch loop is attractive because it makes it easy to instrument the execution of a virtual program. Figure 3 shows how instrumentation can be interposed before and after the dispatch of each virtual instruction. The figure illustrates a generic form of dispatch loop (the shaded rectangle in the lower right) where the instrumentation routines to be called are implemented as function pointers accessible via the vPC. In addition the dispatcher payload is passed to each instrumentation call. The disadvantage of this approach is that the dispatch of the instrumentation is burdened by the overhead of a call through a function pointer. This is not a problem because as larger regions are identified and executed the frequency of dispatch falls by orders of magnitude.

Our strategy for identifying regions of a virtual program requires every thread to execute in one of several "modes". For instance, when generating a trace, a thread will be in *trace generation mode*. Each thread has associated with it a *thread context structure* (tcs) that includes various mode bits as well as the *history list*, which is used to accumulate regions of the virtual program.

3.2 Loading

When a method is first loaded we don't know which parts of it will be executed. Hence, to avoid load time overhead for code that never runs, we adopt a lazy strategy. As each instruction is loaded its first slot is initialized to a shared dispatcher structure. There is one shared dispatcher for each kind of virtual instruction. One instance is shared for all iload instructions, another instance for all iadd instructions, and so on. Thus, the minimal work is done at load time for instructions that never run. On the other hand, a shared dispatcher cannot be used to profile instructions that do execute. Hence, the shared dispatcher is replaced by a new, non-shared, instance of a *block discovery dispatcher* when the postworker of the shared dispatcher runs for the first time. The job of the block discovery dispatcher is to identify new linear blocks[2].

3.3 Linear Block Detection

When the preworker of a block discovery dispatcher executes for the first time, and the thread is *not* already recording a region, the program is about to enter a linear block that has never run before. When this occurs we switch the thread into *block recording mode* by setting a bit in the thread context structure. Figure 3 illustrates the discovery of the linear block of our running example. The postworker called after the execution of each instruction has appended the instruction's payload to the thread's history list. When a branch instruction is encountered by a thread in block recording mode, the history list is used to generate a region body for the linear block. Figure 4 illustrates the situation just after the collection of the linear block. The dispatcher corresponding to the entry point of the linear block has been replaced by a new *linear block dispatcher* whose job will be to search for traces. The linear block dispatcher includes a new payload created from the history list; its body field points to a subroutine-threading-style region body that has been generated for the linear block. Note that linear blocks are not basic blocks because they do not end at labels. If the virtual program later branches

[2] This approach saves the overhead of creating a dispatcher at load time for instructions that never run.

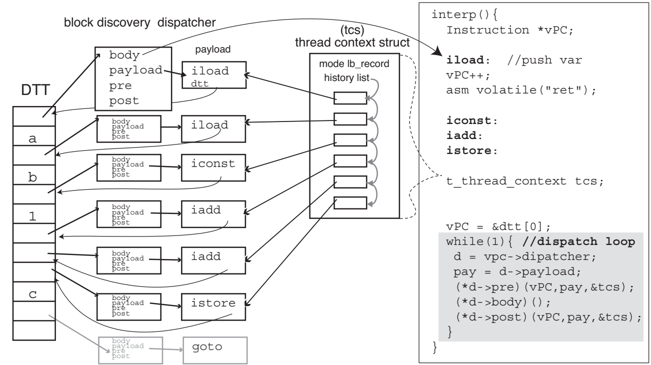

Figure 3. Shows a region of the DTT during block recording mode. The body of each block discovery dispatcher points to the corresponding virtual instruction body (Only the body for the first iload is shown). The dispatcher's payload field points to instances of instruction payload. The thread context struct is shown as tcs.

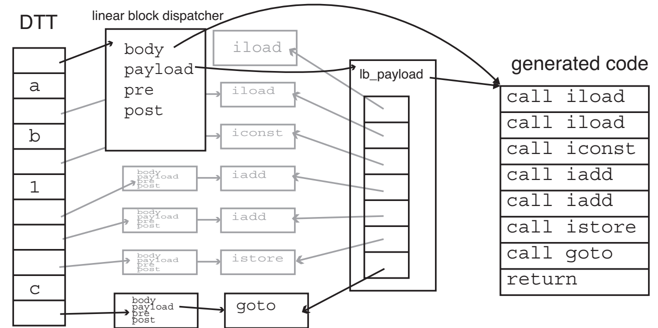

Figure 4. Shows a region of the DTT just after block recording mode has finished.

to a virtual address that happens to be in the middle of a linear block our system will create a new linear block that replicates the tail of the original.

3.4 Trace Selection

The postworker of a linear block dispatcher is called after the last virtual instruction of the block has been dispatched. Since linear blocks end with branches, after executing the last instruction the vPC points to one of the successors of the linear block. If the vPC of the destination is *less* than the vPC of the virtual branch instruction, this is a reverse branch – a likely candidate for the latch of a loop. According to the heuristics developed by Dynamo (see Section 2.2), hot reverse branches are good places to start the search for hot code. Accordingly, when our system detects a reverse branch that has executed 100 times it enters *trace recording mode*[3]. In trace recording mode, much as in linear block recording mode, the postworker adds each linear block to a history list. The situation is very similar to that illustrated in Figure 3, except the history list records linear blocks. Our system, like Dynamo, ends a trace (i) when it reaches a reverse branch or finds a cycle, or (ii) when it contains too many (currently 100) linear blocks. When trace selection ends, a new *trace dispatcher* is created and installed. This is quite similar to Figure 4 apart from the need to support trace exits. The payload of a trace dispatcher includes a table of *trace exit descriptors*, one for each linear block in the trace. Although code

[3] Performance was not sensitive to the specific value so we chose a round number in the range of values used by Dynamo.

could be generated for the trace at this point, we postpone code generation until the trace has run a few times, currently five, in trace training mode[4]. Trace training mode uses a specialized dispatch loop that calls instrumentation before and after dispatching each virtual instruction in the trace. In principle, almost any detail of the virtual machine's state could be recorded. Currently, we record the class of every Java object upon which a virtual method is invoked. When training is complete, code is generated for the trace as illustrated by Figure 5. Before we discuss code generation, we need to describe the runtime of the trace system and especially the operation of trace exits.

3.4.1 Trace Exit Runtime

Trace exits occur when execution diverges from the path collected during trace generation, in other words, when the destination of a virtual branch instruction in the trace is different from what was recorded during trace generation. Generated guard code in the trace detects the divergence and branches to a *trace exit handler*. Generated code in the trace exit handler records which trace exit has occurred in the thread's context structure and then returns to the dispatch loop, which immediately calls the postworker corresponding to the trace. The postworker determines which trace exit occurred by examining the thread context structure. Conceptually, the postworker has only a few things it can do:

1. If the trace exit is still cold, increment a counter in the corresponding trace exit descriptor.
2. Notice that the counter has crossed the hot threshold and arrange to generate a new trace.
3. Notice that a trace already exists at the destination and link the trace exit handler to the new trace.

Regular conditional branches, such as the Java `IF_ICMP`, are quite simple. The branch has only two destinations, one on the trace and the other off. When the trace exit becomes hot a new trace is generated starting with the off-trace destination. Then, the next time the trace exit occurs, the postworker links the trace exit handler to the new trace by rewriting the branch instruction in the trace exit handler to jump directly to the destination trace instead of returning to the dispatch loop. Subsequently, execution stays in the trace cache for both paths of the program.

Multiple destination branches, like method invocation and return, are more complex. When a trace exit originating from a multi-way branch occurs we are faced with two additional challenges. First, profiling multiple destinations is more expensive than just maintaining one counter. Second, when one or more of the possible destinations are also traces, the trace exit handler needs some mechanism to jump to the right one.

The first challenge we essentially punt on. We use a simple counter and trace generate *all* destinations of a hot trace exit that arise. The danger of this strategy is that we could trace generate superfluous cold destinations and waste trace generation time and trace cache memory.

The second challenge concerns the efficient selection of a destination trace to which to link, and the mechanics used to branch there. To choose a destination, we follow the heuristic developed by Dynamo for regular branches – that is, we link to destinations in the order they are encountered. At link time, we rewrite the code in the trace exit handler with code that checks the value of the `vPC`. If it equals the `vPC` of a linked trace, we branch directly to that trace; otherwise we return to the dispatch loop. Because we know the specific values the `vPC` could have, we can hard-wire the comparand in the generated code. In fact, we can generate a sequence of compares checking for two or more destinations. Eventually, a sufficiently long cascade would perform no better than a trip around the dispatch loop. Currently we limit ourselves to two linked destinations per trace exit. This mechanism is similar to a PIC, used to dispatch polymorphic methods, as discussed in Section 2.3.

[4] As almost all the callsites in the SPECjvm98 benchmarks are monomorphic, a smaller number of training runs would have been sufficient but unrealistic.

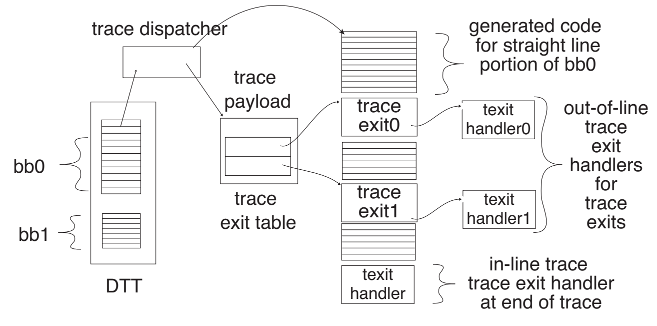

Figure 5. Schematic of a trace

3.5 Generating code for traces

Generating code for a trace is made up of two main tasks, generating the main body of the trace and generating a trace exit handler for each trace exit. After trace selection we have a list of linear blocks that were selected. We will use these to access the virtual instructions making up the trace. After a few training runs we also have fine-grained profiling information on the precise values that occur during the execution of the trace. These values will be used to devirtualize selected virtual method invocations.

3.5.1 Straight line code generation

The body of a trace is made up of straight-line sections of code, corresponding to the body of each linear block, interspersed with trace exits generated from the virtual branches ending each linear block. At this phase of our research we have not invested any effort in generating optimized code for the straight-line portions of a trace. Instead, we implemented a simple one-pass compiler. An important aspect of our design is that it can generate code for every trace before our JIT supports all virtual instructions. Our JIT generates register-to-register code for contiguous sequences of virtual instructions it recognizes. (These include all the conditional branch instructions.) When an unfamiliar virtual instruction is encountered code is generated to flush any temporary values held in registers back to the Java expression stack[5]. Then, a sequence of calls is generated to dispatch the bodies of any uncompilable virtual instructions. This significantly eases development as the compiler can be extended one virtual instruction at a time.

The actual machine code generation is performed using the ccg [19] runtime assembler.

3.5.2 Trace Exits and Trace Exit Handlers

The virtual branch instruction ending each block is compiled into a trace exit. We follow two different strategies for trace exits. The first case, regular conditional branch virtual instructions, are compiled by our JIT into code that performs a compare followed by a conditional branch. PowerPC code for this case appears in Figure 6. The sense of the conditional branch is adjusted so that the branch is always not-taken for the on-trace path. More complex virtual branch instructions, and especially those with multiple destinations, are handled differently. Instead of generating inlined code

[5] Every value assigned to a register by our simple register allocator has a "home" location in the Java expression stack.

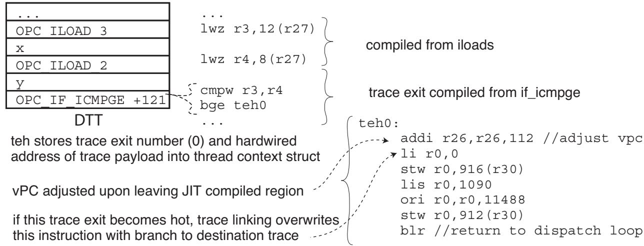

Figure 6. PowerPC code for a portion of a trace region body, showing details of a trace exit and trace exit handler. This code assumes that r26 has been dedicated for the `vPC`.

for the branch we generate a call to the virtual branch body instead. This will have the side effect of setting the `vPC` to the destination of the branch. Since only one destination can be on-trace, and since we know the exact `vPC` value corresponding to it, we then generate a compare immediate of the `vPC` to the hardwired constant value of the on-trace destination. Following the compare we generate a conditional branch to the corresponding trace exit handler. The result is that execution leaves the trace if the `vPC` set by the dispatched body was different from the `vPC` observed during trace generation. Polymorphic method dispatch is handled this way if it cannot be optimized as described in Section 3.5.3.

Trace exit handlers have two further roles not mentioned so far. First, since traces may contain compiled code, it may be necessary to flush values held in registers back to the Java expression stack before returning to regular interpretation. Similarly, it is possible to delay updating values needed only by the interpreter, like the `vPC`, until the end of a section of compiled code. Code is generated for both these purposes in each trace exit handler. For instance, in Figure 6, the trace exit handler adjusts the `vPC`. Second, trace linking is achieved by overwriting code in a trace exit handler. (This is the only situation in which we rewrite code.) To link traces, the tail of the trace exit handler is rewritten to branch to the destination trace rather than return to the dispatch loop.

Most trace exit handlers are reached only when a conditional trace exit is taken. When a trace executes to completion, however, control must initially return to the dispatch loop. To implement this each trace ends with an in-line trace exit handler. Like any other trace exit handler, it may later be linked to its destination trace if one becomes hot.

3.5.3 Trace Optimization

We describe two optimizations here: how loops are handled and how the training data can be used to optimize method invocation.

Inner Loops One property of the trace selection heuristic is that innermost loops of a program are often selected into a single trace with the reverse branch at the end. (This is so because trace generation starts at the target of reverse branches and ends whenever it reaches a reverse branch. Note that there may be many branches, including calls and returns, along the way.) Thus, when the trace is generated the loop will be obvious because the trace will end with a virtual branch back to its beginning. So far we exploit this information only so far as to compile the last trace exit in a trace to a conditional branch back to the head of the trace.

Virtual Method Invocation When a trace executes, if the class of the invoked-upon object is different from when the trace was generated, a trace exit must occur. At trace generation time we know the on-trace destination of each call. From the training profile we know the class of each invoked-upon object. Thus, we can easily generate a *virtual invoke guard* that branches to the trace exit handler if the class of the object on top of the Java runtime stack is not the same as recorded during training. Then, we can generate code to perform a faster, stripped down version of method invocation. The savings are primarily the work associated with looking up the destination given the class of the receiver. The virtual guard is an example of a trace exit that guards a speculative optimization [12].

Inlining Traces are agnostic towards method invocation and return, treating them like any other multiple-destination virtual branch instructions. However, when a return corresponds to an invoke in the same trace the trace generator can sometimes remove almost all method invocation overhead. Consider when the code between a method invocation and the matching return is relatively simple, for instance, it does not touch the callee's stack frame (other than the expression stack), it cannot throw and it makes no method invocations. Then, no invoke is necessary and the only method invocation overhead that remains is the virtual invoke guard. If the inlined method body contains any trace exits the situation is slightly more complex. In this case, in order to prepare for a return somewhere off-trace, the trace exit handlers for the trace exits in the inlined code must modify the runtime stack exactly as the (optimized away) invoke would have done. Currently our implementation can inline only to a depth of one.

3.6 Polymorphic bytecodes

So far we have implemented our ideas in a Java virtual machine. However, we expect that many of the techniques will be useful in other virtual machines as well. For instance, languages such as TCL or JavaScript define polymorphic virtual arithmetic instructions. An example would be ADD, which adds the two values on the top of the expression stack. Each time it is dispatched ADD must check the type of its inputs and perform the correct type of arithmetic. This is similar to polymorphic method invocation.

We believe the same profiling infrastructure that we use to optimize monomorphic callsites in Java can be used to improve polymorphic arithmetic bytecodes. Whereas the destination of a Java method invocation depends only upon the type of the invoked upon object, the operation carried out by a polymorphic virtual instruction may depend on the type of *each* input. For instance, suppose that an ADD in TCL is effectively monomorphic. We could generate two virtual guards, one for each input, to check that the type of the input is the same as observed during training and trace exit if it differs. Then, we could dispatch a type-specialized version of the instruction (integer ADD, float ADD, string ADD, etc.) and/or generate specialized code for common cases.

3.7 Other implementation details

Our system, as described above, generates code that coexists with virtual instruction bodies written in C. Consequently, our generated code sometimes must know the stack layout and register allocation chosen by the compiler for certain values used by the virtual instruction bodies. For heavily used interpreter variables, like the `vPC`, the obvious solution is to use gcc compiler extensions to assign the variable to a dedicated register.

Our use of a dispatch loop similar to Figure 3 in conjunction with ending virtual bodies with inlined assembler return instructions results in a control flow graph that is not apparent to the compiler. This is because the optimizer cannot know that control flows from the inlined return instruction back to the dispatch loop. Similarly, the optimizer cannot know that control can flow from the function pointer call in the dispatch loop to any body. We insert computed goto's that are never actually executed to simulate the missing edges.

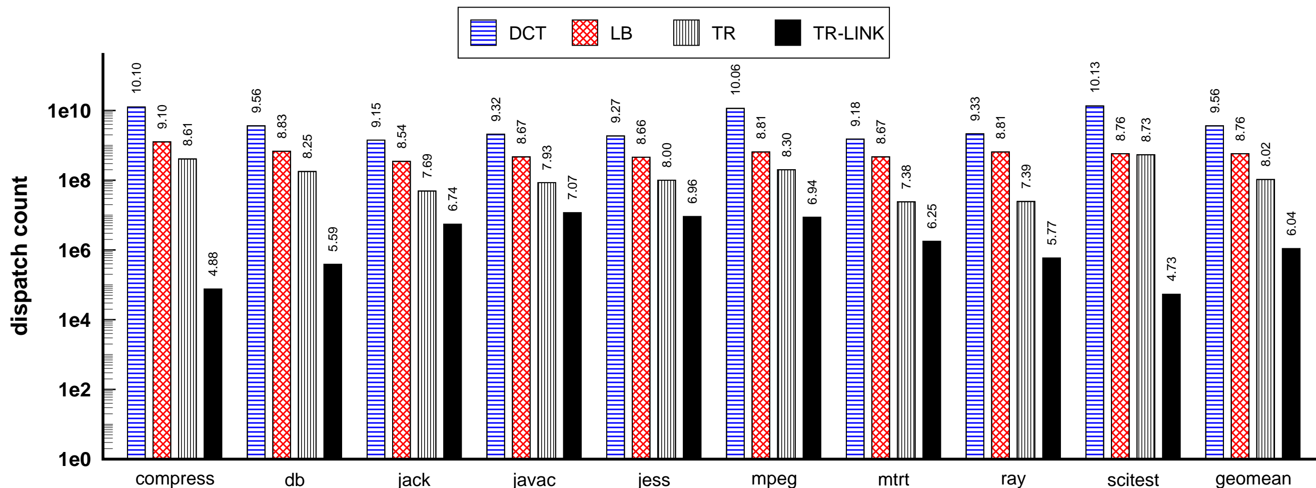

Figure 7. Number of dispatches executed vs region shape. The y-axis has a logarithmic scale. Numbers above bars are $\log_{10}$ of the dispatch count.

3.8 Packaging and portability

An attractive starting point when building an interpreter from scratch would be to use direct call threading with bodies packaged as nested functions (an extension provided by gcc and other C compilers). This would allow a portable implementation, convenient debugging (e.g., logging in the dispatch loop), and a forward path to dynamic compilation. It may be necessary to differentiate platforms into "primary" targets (i.e., those supported by our trace-oriented JIT) and "secondary" targets supported only by portable direct call threading. In Section 4.2 we report that our lightweight approach to direct call threading has about the same performance as switched interpretation, but we have not yet investigated the performance of nested functions.

When retrofitting our techniques into a direct threaded interpreter, conditional compilation can be used to allow bodies to end with either a computed `goto` or an inline assembler "`ret`". Thus, secondary platforms can continue to use direct threading.

4. Experimental Results

In this section we show how Yeti steadily improves in performance as we extend the size of region bodies. We prototyped Yeti in a Java VM (rather than a language that does not have a JIT) to allow comparisons of well-known benchmarks against other high-quality implementations.

To evaluate the effectiveness of our system we need to examine performance from three perspectives. First, we show that almost all execution comes from the trace cache. Second, we show the incremental effect on execution time of each step in the development of Yeti. Third, we compare the overall performance of Yeti against other Java interpreter and JIT implementations, including SableVM, a version of JamVM modified to use our earlier subroutine threading technique[2], and Sun's optimizing HotSpot Java virtual machine.

Table 1 briefly describes each SPECjvm98 benchmark [21]. We also report data for `scimark`, a typical scientific program. Below we report performance relative to the unmodified JamVM 1.3.3, so the raw elapsed time for each benchmark also appears in Table 1, along with the raw elapsed time of our best-performing version of Yeti which includes our simple JIT.

Table 1. SPECjvm98 benchmarks including elapsed time for baseline JamVM 1.3.3 (i.e., without any of our modifications), Yeti and Sun HotSpot 1.05.0_6_64.

Benchmark	Description	Elapsed Time (sec)		
		JamVM	Yeti	HotSpot
compress	Lempel-Ziv	98	44	8.0
db	Database functions	56	35	23
jack	Parser generator	22	14	5.4
javac	JDK 1.0.2	33	24	9.9
jess	Expert Shell System	29	19	4.4
mpeg	read MPEG-3	87	36	4.6
mtrt	Two thread raytracer	30	25	2.1
raytrace	raytracer renderer	29	17	2.3
scimark	FFT, SOR,LU, 'large'	145	58	16

We present data obtained by running various modifications to JamVM version 1.3.3 built with gcc 4.0.1. All our data was collected on a dual CPU 2 GHz PPC970 processor with 512 MB of memory running Apple OSX 10.4. Performance is reported as the average of three measurements of elapsed time, as printed by the `time` command.

4.1 Effect of region shape on region dispatch count

For a JIT to be effective, execution must spend most of its time in compiled code. For `jack`, traces account for 99.3% of virtual instructions executed. For all the remaining benchmarks, traces account for 99.9% or more. A remaining concern is how often execution enters and leaves the trace cache. In our system, regions of generated code are called from dispatch loops like the one illustrated by Figure 3. In this section, we report how many iterations of the dispatch loops occur during the execution of each benchmark. Figure 7 shows how direct call threading (DCT) compares to linear blocks (LB), traces with no linking (TR) and linked traces (TR-LINK). Note the y-axis has a logarithmic scale.

DCT dispatches each virtual instruction independently, so the DCT bars on Figure 7 report how many virtual instructions were executed. For each benchmark, the ratio of DCT to LB shows the dynamic average linear block length (e.g., for compress the average linear block length is $10^{10.1}/10^{9.1} = 10^1 = 10$). As expected,

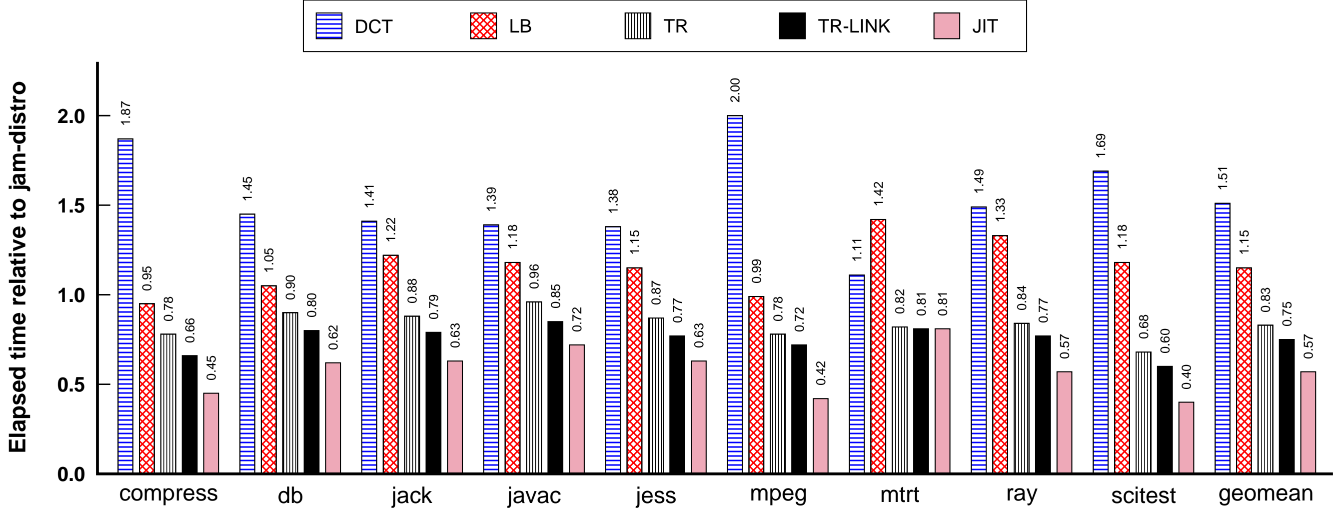

Figure 8. Performance of Yeti relative to unmodified JamVM-1.3.3 (direct threaded) running the SPECjvm98 benchmarks.

the scientific benchmarks have longer linear blocks. For instance, the average block in `scitest` has about 20 virtual instructions whereas `javac`, `jess` and `jack` average about 4 instructions. Comparing the geometric mean across benchmarks, we see that LB reduces the number of dispatches relative to DCT by a factor of 6.3.

Even without trace linking, a trace executes about 10 times more virtual instructions per dispatch than a LB. This can be calculated from Figure 7 as the ratio of LB to TR. This shows that traces do predict the path taken through the program. The improvement can be dramatic. For instance, while running TR, `javac` executes about 22 virtual instructions per trace dispatch, on average. This is much longer than its dynamic average linear block length of 4 virtual instructions.

TR-LINK makes the greatest contribution, reducing the number of times execution leaves the trace cache by between one and 3.7 *orders of magnitude*. The reason TR-LINK is so effective is that it links traces together around loops.

Although this data shows that execution is overwhelmingly from the trace cache it gives no indication of how effectively code cache memory is being used by the traces. A thorough treatment of this, like the one done by Bruening and Duesterwald [3], remains future work. Nevertheless, we can relate a few anecdotes based on data that our profiling system collects. We observe that for an entire run of the `compress` benchmark all generated traces contain only 60% of the virtual instructions contained in all loaded methods. This is a good result for traces, suggesting that a trace-based JIT needs to compile fewer virtual instructions than a method-based JIT. On the other hand, for `javac` we find that the traces bloat – almost eight *times* as many virtual instructions appear in traces than are contained in the loaded methods. Improvements to our trace selection heuristic, perhaps adopting the suggestions of Hiniker et al [14], are future work.

4.2 Effect of region shape on performance

Figure 8 shows how performance varies as differently shaped regions of the virtual program are executed. The figure shows elapsed time relative to the unmodified JamVM distribution, which uses direct-threaded dispatch. The raw performance of unmodified JamVM is given in Table 1. The first four bars in each cluster are the same as Figure 7. The fifth bar, JIT, gives the performance of Yeti with JIT enabled.

The simplest technique, direct call threading (DCT) is slower than direct threading by about 50%. DCT is a baseline, in the sense that it burdens the execution of every virtual instruction with the overhead of the dispatch loop. Not shown in the figure is switch dispatch, for which the geometric mean elapsed time across all the benchmarks is within 1% of DCT.

"Linear blocks" (LB) runs roughly 30% faster than DCT, as expected given the reduction in dispatch count seen in Figure 7, and is comparable to direct threading in several cases. LB discovers and generates code at run time that is very similar to that generated by subroutine threading (SUB) at load time, so it is interesting to compare the two techniques. The geometric mean across the benchmarks of LB is about 43% slower than SUB (not shown). The difference between them is the cost of instrumentation and dynamic detection. Although SUB is an efficient interpreter dispatch technique, it is difficult to extend to dynamic regions, primarily because it is hard to add and remove the necessary profiling.

Just as LB reduces dispatch and performs better than DCT, so traces (TR) further reduce dispatch, running 38% faster than LB. In addition to fewer dispatches, traces also use a lighter-weight dispatch loop with no further need for profiling. Although TR-LINK dramatically reduces the number of dispatches, the performance gain is relatively smaller because the specialized dispatch loop used for traces is less expensive.

Comparing to other interpreters, we note that TR-LINK outperforms SUB on the geometric mean of the benchmarks by about 6%, and SableVM 1.1.8 by about 4%. Thus, TR-LINK more than makes up for the profiling overhead required to identify and generate traces. The advantage of TR-LINK over SUB is that virtual branch instructions are converted into trace exits, where they are exposed to the hardware branch predictors.

For all benchmarks, performance improves as region bodies become longer, that is, LB performs better than DCT, TR performs better than LB, etc. This shows that our approach allows us to improve performance by investing in better region selection.

4.2.1 JIT Compiled traces

The rightmost bar in each cluster of Figure 8 shows the performance of our best-performing version of Yeti (JIT). Despite supporting only 50 integer and object virtual instructions, our trace JIT improves the performance of integer programs such as `compress` significantly. With our most ambitious optimization, of virtual

method invocation, JIT improved the performance of `raytrace` by about 35% over TR-LINK. `Raytrace` is written in an object-oriented style with many small methods invoked to access object fields. Hence, even though it is a floating-point benchmark, it is greatly improved by devirtualizing and inlining the accessor methods. Comparing geometric means, we see that our trace-oriented JIT is roughly 32% faster than TR-LINK.

Our current JIT runs the SPECjvm98 benchmarks 4.3 times slower than Sun's optimizing HotSpot compiler. Results range from 1.5 times slower for `db`, to 12 times slower for `mtrt`. Not surprisingly, we do worse on floating-point intensive benchmarks since we do not yet compile the float bytecodes.

5. Related Work

Other related work has been discussed in Section 2. Here, we discuss one closely-related system, contrasting it to Yeti.

Hotpath also extends JamVM to be a trace-oriented mixed-mode system [12]. Its profiling system, similar to those used by many method-based JITs, is loosely coupled with the interpreter. Hotpath focuses on traces starting at loop headers and does not compile traces not in loops. Thus, it does not approach trace linking as we do, but rather "merges" traces that originate from side exits leading back to loop headers, allowing it to compile loop nests. They model traces using a Single Static Assignment (SSA) representation that exploits the constrained flow of control present in traces. This both simplifies their construction of SSA and allows very efficient optimization. Their experimental results show good speedup, within a factor of two of Sun's HotSpot, for scientific style loop nests such as those in LU, SOR and Linpack, and more modest speedup, around a factor of two over interpretation, for FFT. Although it is difficult to compare directly, we note that Yeti achieves a speedup of 2.6 relative to JamVM on the FFT test in `scimark`.

Hotpath has concentrated on how traces should be optimized whereas we have concentrated on how a trace-oriented interpreter and JIT should be integrated. The optimization techniques they describe seem complementary to the overall architecture we propose.

6. Conclusions and future work

Yeti is an architecture for a virtual machine interpreter that facilitates extension to a trace-based mixed-mode JIT compiler. By taking a step back from high-performance dispatch techniques to direct call threading we achieve two benefits. The first is that existing bodies can be reused by generated code, so that compiler support for virtual instructions can be added one by one. The second benefit is that it is easy to add instrumentation, allowing us to discover hot regions of the program and to install new region bodies. The cost of this flexibility is increased dispatch overhead. We have shown that by generating larger region bodies, the frequency of dispatch is reduced significantly leading to better performance. Linked traces run 33% faster than a direct threaded interpreter. Investing the modest additional effort to generate non-optimized code for roughly 50 integer and object bytecodes within traces allows Yeti to run nearly twice as fast as direct threading. This demonstrates that it is indeed possible to achieve gradual, but significant, performance gains through gradual development of a JIT.

Substantial additional performance gains are possible by extending the JIT to handle more types of instructions such as the floating-point bytecodes, and by applying classical optimizations such as common subexpression elimination. Far more interesting, however, is the opportunity to apply dynamic and speculative optimizations based on the profiling data that we already collect. The technique we describe for optimizing virtual dispatch in Section 3.5.3 could be applied to guard various speculations. For example, it could be used to optimize virtual instructions that must accept arguments of varying type in languages like Python or JavaScript. Finally, just as linear blocks are collected into traces, so traces can be collected into larger units for optimization.

The techniques we applied in Yeti are not specific to Java. A system based on our architecture can gradually bring the benefits of mixed-mode JIT compilation to other interpreted languages.

Acknowledgments

The authors would like to thank a number of individuals and organizations for their support. We would like to thank Marc Berndl and Benjamin Vitale for technical discussions and comments on this research. Thanks also to Patric Doyle, Marin Litoiu, Derek Inglis, Mark Stoodley and many others at IBM and the University of Toronto for many interesting discussions, and to the anonymous reviewers for their helpful comments on the presentation of this work. Funding for this research was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the IBM Centre for Advanced Studies (CAS).

IBM Legal Statement

IBM and PowerPC are registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

The opinions expressed in this article are those of the authors and do not necessarily represent those of International Business Machines Corporation or any of its affiliates.

Neither International Business Machines Corporation nor any of its affiliates assume any responsibility or liability in respect of any results obtained by implementing any recommendations contained in this article. Implementation of any such recommendations is entirely at the implementor's risk.

Publication of this article, including any recommendations contained in this article, does not confer any license or other right under any patent or patent application owned by International Business Machines Corporation or any of its affiliates.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proc. of the ACM SIGPLAN 2000 Conf. on Prog. Language Design and Impl.*, pages 1–12, Jun. 2000.

[2] M. Berndl, B. Vitale, M. Zaleski, and A. D. Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *Proc. of the 3rd Intl. Symp. on Code Generation and Optimization*, pages 15–26, Mar. 2005.

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

[4] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proc. of the 1st Intl. Symp. on Code Generation and Optimization*, pages 265–275, Mar. 2003.

[5] W.-K. Chen, S. Lerner, R. Chaiken, and D. Gillies. Mojo: A dynamic optimization system. In *Proc. of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, Dec. 2000.

[6] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proc. of the 1st Intl. Symp. on Code Generation and Optimization*, pages 15–24, Mar. 2003.

[7] K. Driesen. *Efficient Polymorphic Calls*. Klumer Academic Publishers, 2001.

[8] E. Duesterwald and V. Bala. Software profiling for hot path prediction: less is more. *ACM SIGPLAN Notices*, 35(11):202–211, 2000.

[9] M. A. Ertl. Stack caching for interpreters. In *Proc. of the ACM SIGPLAN 1995 Conf. on Prog. Language Design and Impl.*, pages 315–327, June 1995.

[10] M. A. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proc. of the ACM SIGPLAN 2003 Conf. on Prog. Language Design and Impl.*, pages 278–288, June 2003.

[11] E. Gagnon and L. Hendren. Effective inline threading of Java bytecode using preparation sequences. In *Proc. of the 12th Intl. Conf. on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 170–184. Springer, Apr. 2003.

[12] A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proc. of the 2nd Intl. Conf. on Virtual Execution Environments*, pages 144–153, 2006.

[13] K. Hazelwood and D. Grove. Adaptive online context-sensitive inlining. In *Proc. of the 1st Intl. Symp. on Code Generation and Optimization*, pages 253–264, Mar. 2003.

[14] D. Hiniker, K. Hazelwood, and M. D. Smith. Improving region selection in dynamic optimization systems. In *Proc. of the 38th Intl. Symp. on Microarchitecture*, pages 141–154, Nov. 2005.

[15] U. Hölzle. *Adaptive Optimization For Self:Reconciling High Performance With Exploratory Programming*. PhD thesis, Stanford University, 1994.

[16] R. Lougher. JamVM [online]. Available from: `http://jamvm.sourceforge.net/`.

[17] M. Paleczny, C. Vick, and C. Click. The Java HotSpot^{TM} server compiler. In *Proc. of the USENIX Java Virtual Machine Research and Technology Symposium*, pages 1–12, Apr. 2001.

[18] I. Pechtchanski and V. Sarkar. Dynamic optimistic interprocedural analysis: A framework and an application. In *Proc. of the 16th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 195–210, Oct. 2001.

[19] I. Piumarta. Ccg: A tool for writing dynamic code generators. In *OOPSLA'99 Workshop on simplicity, performance and portability in virtual machine design*, Nov. 1999.

[20] I. Piumarta and F. Riccardi. Optimizing direct-threaded code by selective inlining. In *Proc. of the ACM SIGPLAN 1998 Conf. on Prog. Language Design and Impl.*, pages 291–300, June 1998.

[21] SPECjvm98 benchmarks [online]. 1998. Available from: `http://www.spec.org/osg/jvm98/`.

[22] D. Sugalski. Implementing an interpreter [online]. Available from: `http://www.sidhe.org/%7Edan/presentations/Parrot%20Implementation.ppt`. Notes for slide 21.

[23] 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), Feb. 2000.

[24] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for dynamic compilers. *ACM Trans. Program. Lang. Syst.*, 28(1):134–174, 2006.

[25] V. Sundaresan, D. Maier, P. Ramarao, and M. Stoodley. Experiences with multi-threading and dynamic class loading in a Java just-in-time compiler. In *Proc. of the 4th Intl. Symp. on Code Generation and Optimization*, pages 87–97, Mar. 2006.

[26] D. Ungar, R. B. Smith, C. Chambers, and U. Hölzle. Object, message, and performance: how they coexist in Self. *IEEE-COMPUTER*, 25(10):53–64, Oct. 1992.

[27] J. Whaley. Partial method compilation using dynamic profile information. In *Proc. of the 16th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 166–179, Oct. 2001.