Abstract

This document describes the progress of the project on porting existing YETI [7] for x86 architecture. It shows what changes were made in the existing YETI project, gives short literature survey with some details in YETI design and describes what results were obtained on the SPECjvm98 benchmark.

1 Introduction

The main goal of the project is to obtain high efficiency for the Java Virtual Machine Interpreter on x86 machines using the same techniques as were used in YETI [7]. The main reason to do this is that we want to make YETI applicable for a much larger number of existing computers (x86 not only PowerPC). In order to achieve this goal a revision of the existing code was made to find out all platform-dependent parts. These parts were separated from the rest of the code and were rewritten using the same run-time assembler called ccg [5]. The second part of the porting process was the bug fixing. We found several bugs that were algorithmic errors and this part was probably the most time consuming part of the work.
2 Motivation

Interpreters for different Virtual Machines (VM) are usually designed to achieve high efficiency. One of the main problems in obtaining this high performance are indirect branch mispredictions. Branch Target Buffers (BTB) are widely used for indirect branch prediction, but their accuracy can be low (2%–50% as in [3]). One of the way to attack this problem is using a dispatch technique called context threading [2] that improves branch prediction and performance by aligning hardware and virtual machine state. In this technique linear VM’s instructions are dispatched with native calls and returns. Conversion from virtual branching instructions to native branches is also used in this technique. To achieve a better branch prediction a new structure called the Context Threading Table (CTT) was added to the interpreter architecture. It contains a sequence of native call instructions and each call dispatches the body of its virtual instruction. Each non-branching opcode instruction now ends with a native return, while branch opcodes ends with an indirect jump. The call of subroutine costs two control transfers, while direct threading costs only one, but a large number of mispredicted branches are avoided. After using subroutine threading, control flow is still hidden from the hardware.

Two approaches can be used to address this problem:
- branch replication replicate an indirect branch instruction in the CTT immediately after the call to the branching opcode body. Very simple, but we have now 3 hardware control transfers and do not use all the power of the conditional branch predictor.
- branch inlining convert indirect branches into direct ones where possible. Code size is increased, but not significantly.

3 Interpreter Background

Our work was based mostly on YETI [7], context threading [2] and Dynamo [1] papers. So, we will describe these three papers in more detail. Some papers that were trying to solve the same problem of an effective interpreter constructing, we will describe briefly.
3.1 The YETI

The YETI paper [7] tries to solve a problem of a large performance gap between efficient interpreters and mixed mode systems that include a just-in-time (JIT) compiler. One of the ideas is to make the JIT more closely integrated with the interpreter. The second idea is that compiling of cold code could cause not only wasting compile time, but might need to solve problems of late binding and inefficient code generating for cold regions (that could become hot later).

So, to solve these problems:
1. Unit of compilation should have dynamical shape (region body)
2. Hot regions need a way to be compiled and linked effectively
3. Virtual bodies should be callable routines

YETI (gradually Extensible Trace Interpreter) has 5 main phases:
- Repackaging for all virtual instructions to make them callable
- Linear blocks creation
- Trace (linear blocks sequences) making
- Trace linking
- Trace compiling (currently 50 integer virtual instructions and generating calls for the rest)

YETI is based on the JamVM Java virtual Machine. YETI runs a program by processing every instruction in a dispatch loop, where region bodies, linear blocks, then traces and linked traces are initially acquired. Trace selection in YETI is based on the idea from Dynamo [1] that hot reverse branches are good places to start the search for hot code. One very good property of this selection technique is that innermost loops are always selected in one trace.

The actual machine code generation is made by the ccg [5] run-time assembler. To make coexistence of the generated code with virtual instructions bodies (written in C) possible a special trick was used. Computed goto’s that are never actually executed were added to say the optimizer that there was a control-flow back to the dispatch loop.

3.2 The Context Threading and Dynamo

The second main paper is the context threading one [2] (it is also a basis for YETI [7]). It tries to solve a problem of indirect branch mispredictions that can be a bottleneck in achieving high performance. One of the way to attack this problem is using a dispatch technique called context threading that improves branch prediction and performance by aligning hardware and
virtual machine state. More details are in Motivation chapter.
The trace approach that was used in YETI was originally described in Dy-
namo paper [1]. This paper is about Dynamo—a dynamic optimization
system and the main idea was to focus on few specific optimizations that
are likely to manifest themselves only at runtime and do not work in static
variants at all. Initially, the authors made the decision that it is better to
optimize very hot parts of code that need to be acquired by some algorithm.
These sequential parts of code are called traces and their optimized variant,
generated by Dynamo, is called a fragment. Then the authors made the con-
ventional assumption that the majority of the application execution time is
spent on a very small part of its code; however some current works say that
it is not always so (i.e. gcc test in every SPEC benchmark). Using this as-
sumption the authors devoted all their efforts to optimizing the small parts
of very hot code collected in traces. Their algorithm (MRET most recently
executed tail) for collecting traces is simple and thus easy to implement, but
it has a numerous performance disadvantages, so they propose to optimize
it for current needs.
The authors used a few optimization techniques. These include deletion of
unconditional direct branches, partial deletion of branch-and-link branches
with side-effects, predicting indirect branches and replacing them with direct
conditional ones and fragment linking.

3.3 Inlining and Superinstructions

One of the ways to solve the problem of branch misprediction is selective in-
lining described in Piumarta & Riccardi paper [6]. The authors’ main idea
was to optimize common byte code sequences by transforming them into
bigger single macro opcodes. This technique is especially efficient for simple
sequences of the byte codes like 3 + 4. Determining the set of common byte
codes is not difficult, because the virtual machine can be easily instrumented
to collect execution traces and appropriate analysis can be done offline. In-
lining can be done by dynamically rewriting opcode sequences that were
chosen before.
Ertl and Gregg [3] showed later that replicating together with a number of
existing superinstruction techniques could eliminate mispredictions signifi-
cantly.
Ertl and Gregg continued to develop the superinstruction idea and proposed
a combination of stack caching with dynamic superinstructions [4].
4 Solution in detail

4.1 DTT vs CTT

As shown in Figure 1, a virtual program is represented by a sequence of virtual operations that can be loaded into the interpreter by constructing a list of addresses—each address for each virtual instruction in the program. This table is called the **Direct Threading Table** (DTT) and its locations are called slots. The interpreter should have a virtual program counter (vPC) to determine the next virtual instruction to be executed. We start interpretation by initializing vPC to the first slot in our DTT table and then we change control by executing `goto *vPC++`. The problem is that we’ll get an indirect branch from this goto in the generated code and this could cause branch mispredictions that are very expensive on modern CPUs.

One of the possible modifications of direct threading is called **subroutine threading**. In subroutine threading every virtual instruction has a body that can be called (Figure 2). Modern microprocessors contain specialized hardware to improve performance of call and return (i.e., address stack that predicts the destination of the return to be the instruction following the corresponding call; call has a single destination when the code is generated, no prediction is needed). So, we have 2 control flow transfers (more than 5...
with DTT), but we could get a huge benefit by eliminating unpredictable branches. The region of generated call instructions is called Context Threading Table or CTT. The DTT is modified such that the first slot for every virtual instruction points to the corresponding CTT slot with the call. Execution now can come back and forth between the code in the CTT and the virtual instruction bodies.

Generating the code at load time in such a way may be inefficient, especially for the code that is never executed. So, CTT is generated only for parts of the program that are known to execute (using profiling).

### 4.2 Inlining

One of the way to optimize the dispatch process is to eliminate it using selective inlining. This method is very attractive in general, because it allows the interpreter to avoid dispatch overhead completely. But the load overhead could be sometimes unpredictable, because of the serious code size growth. We use this technique for handling virtual branches. Branch replication is attractive, because it is simple and produces the desired context with a minimum of replicated instructions. All branches except for indirect virtual branches and exceptions are fully inlined in the CTT.
4.3 Trace Exits and Trace Exits Handlers

Trace exits occur when execution comes from the path that is different from
the one, we had during trace selection. We generate the guard code in the
trace that detects this situation and transfer control to a trace exit handler.
The code in the trace exit handler increments the counter for the trace exit
that just happened and returns to the main dispatch loop.
The postworker can use the context structure and make 3 possible decisions:
– If the trace exit is cold continue profiling it
– If certain threshold has been reached generate a new trace
– If a trace already exists that just link the trace exit handler with this
trace.
The virtual branch instruction ending each block is compiled into a trace
exit. There two different cases for generating trace exits. The first case,
regular conditional branch, is compiled into compare followed by a condi-
tional branch. The branch is always not-taken for the on-trace path. More
complex case is the case of multi-destination branches. For them we do not
make inlining but generate a call to the virtual body instead. We also gen-
erate a check (compare of the vPC to the hardwired constant of the on-trace
destination) whether we are still on-trace or not.
Trace linking is achieved by overwriting code in the trace handler the only
case when we overwrite code. To make linking, the tail of the trace exit
handler is rewritten to branch to the destination trace rather than return
to the dispatch loop. One special case should be processed when a trace
executes to completion, because we must return to the dispatch loop. To
implement this each trace ends with an in-line trace exit handler.

5 Code Generation

```
movl $bb_num, (%ebp - TrExitOffset)
movl Strace, (%ebp - TrExitTrOffset)
ret
```

Figure 3: Trace Exit
There are two main differences in generating code for x86 compared with PowerPC. One obvious difference is that another set of assembly instructions are used in Intel architecture, so we need to change all PowerPC operations by the equivalent x86 ones. Currently we need to generate code only for different kinds of trace exits, so we were interested in a small subset of operations: `movl`, `cmpl`, `jne`, `ret` and `jmp`.

The second difference is the impossibility of using registers to save information about the current exit. In PowerPC a register with fixed number (currently r30) was used to save both trace exit number and address of trace payload into thread context structure. But we can not use registers for the x86 case, so the solution here is to use memory: `%ebp` register plus offset. The exact value of this offset should be defined manually by making one execution of YETI.

As an example of the generated code, Figure 3 shows the code template that should be made for every trace exit. In this code we put in the memory stack both trace exit number and address of trace payload descriptor using `movl` operations and then make return to a dispatch loop. Similar code is generated in the trace exit handlers.

As was described in YETI paper [7], trace exit handlers were also used to make linking by overwriting code. This was probably the hardest place to port, because it was not easy to debug the generated code that could be changed. In Figure 4 there is an example of the code that can appear after the overwriting procedure. This code checks whether the execution should be continued in the current trace (first line - `cmpl`). If the current VPC in not on-trace then we take conditional branch to the code that saves exit information.

```
cmpl $0x819e028, 0xffffffff(%ebp)
jne 0xab05c52a
jmp 0xaba6201b
movl $0x0, 0xffffffff(%ebp)
movl $0x8151400, 0xffffffff(%ebp)
ret
```

Figure 4: Regenerated Code for Trace Exit
attributes (identical to the normal trace exit). In the opposite case we follow
the jump at line 3 (Figure 4) to the entry point of the destination trace.
Future work on porting YETI to x86 platform will need to generate a code
for a large number of java virtual instructions, and, hence, much more Intel
instructions should be used.

6 Bug fixing

Bug fixing was not the direct goal of the current project, but we expected
events and, as usual, they took much more time than you have planned to
spend working on them. YETI has some internal debugging tools (special
prints and variables) that were very useful while debugging and bug fixing.
We found a small group of simple bugs (i.e. absence of checks and asserts)
and two algorithmic errors.
The most interesting error that was found was the code overwriting for
the case of trace execution for completion (after which we should initially
return to the despatch loop). For this case each trace ends with an in-
line trace exit handler. It can be linked to its destination trace like any
other trace exit handler, but it is not multiple case branch, so it should be
processed differently from all other trace exit handlers that can have many
destinations. Linking this special case like multiple branch causes the code
corruption: Segmentation Fault or Illegal Instruction error. The debugging
process for this error was slowed down at the beginning, because of the
new Linux security feature that moved procedure stacks. As the result, the
execution was nondeterministic.
The second group of problems was connected with gcc compiler. One of
them was known before and predictable - gcc did not ”see” control flow
when we executed generated code by calling the dispatch body; the second
one is still unclear - compiler reused our program’s local variables (saved
new values there). The first one was solved by adding pseudo-call that is
never executed, but it is impossible to prove this fact statically. The second
was solved by using additional local variables.

7 Experimental Results

Of course, we are interested in evaluating the effectiveness of our approach.
We want to show both incremental effect of each step in the development
(Figure 5) and relative performance compared with other Java interpreters
(JamVM in this paper - Figure 6).
Figure 5: SPECjvm98 Benchmark Results

Data was collected by running different modifications of JamVM 1.3.3 built with gcc 3.4.6. We used Intel Pentium4 Core Duo CPU 2.4 GHz (L1 cache - 32Kb, L2 cache - 4 MB) and 2GB of memory running on RedHat Linux 3.4.6-2.

7.1 Results Discussion

In Figure 5 we show results that were obtained for different steps in development of Yeti. First column is just simple interpreter (Direct Call Threading - DCT) without any advance features: no regions or traces. It was surprisingly much slower (relative to distro) than the same execution results for PowerPC. The geometrical mean is more than 3 vs 1.5 for PowerPC. Next column is Yeti variant that has regions with basic blocks, it’s not as slow
Figure 6: SPECjvm98 Benchmark Results - Y eti with traces and linking vs JamVM

as the DCT, but is still slower than JamVM - distro. Third column is the current most advanced version of Y eti for x86 (regions, traces and linking). The last column is our baseline - JamVM - distro.

In Figure 6 we showed last two columns in a better scale. One test hello is just a classical "HelloWorld" test that shows our overhead for almost empty test without any hot cycles. We expected that the results will be worse for x86 than for PowerPC, currently we have got geometrical mean 0.84 vs 0.75 [7]. There can be several reasons for this, probably the main one is that modern x86 architecture is better in branch prediction than PowerPC, so the speedup is lower. Of course, we also run this tests under different OSes. We did not make any special analysis or tuning for x86 and just port our approach as is. So, it can be better solutions for x86 architecture than we
7.2 Future Work

Currently we are going to work on adding the last step in Yeti project for x86 - code generation for Java virtual instructions. It was done for about 50 integer instructions on PowerPC and we hope to do at least a part of this job for x86. This is the most architecture - dependent code in the system and needs good testing cases to cover different uses of every virtual instruction you are going to process. Nevertheless, we do not expect that this will help Yeti to become as powerful as modern JITs, because we do not have any kind of code optimizer.

References


