Journal of Instruction-Level Parallelism 5 (2003) 1-25 Submitted 07/01; published 11/03

The Structure and Performance of *Efficient* Interpreters

M. Anton Ertl ANTON@MIPS.COMPLANG.TUWIEN.AC.AT
Institut für Computersprachen, TU Wien, Argentinierstraße 8, A-1040 Wien, Austria

David Gregg[*] DAVID.GREGG@CS.TCD.IE
Department of Computer Science, Trinity College, Dublin 2, Ireland

Abstract

Interpreters designed for high general-purpose performance typically perform a large number of indirect branches (3.2%–13% of all executed instructions in our benchmarks). These branches consume more than half of the run-time in a number of configurations we simulated. We evaluate how accurate various existing and proposed branch prediction schemes are on a number of interpreters, how the mispredictions affect the performance of the interpreters and how two different interpreter implementation techniques perform with various branch predictors. We also suggest various ways in which hardware designers, C compiler writers, and interpreter writers can improve the performance of interpreters.

1. Introduction

Several advantages make interpreters attractive:

- Ease of implementation.
- Portability.
- Fast edit-compile-run cycle.

Run-time efficiency is an important secondary requirement in many interpretive language implementations. Romer et al. [1] studied the performance of several interpreters (MIPSI, Java, Perl, Tcl). They did not identify performance issues common to all interpreters, or particularly striking results for specific interpreters and concluded that the performance of interpreters should be improved through software means instead of through specialized hardware.

So, in this paper[1], we study four interpreters that have been designed for good general-purpose performance (we call such interpreters *efficient interpreters* in the rest of the paper). We find that they perform an exceptionally high number of indirect branches: each of them has a higher proportion of indirect branches than any of the benchmarks used by researchers on indirect branch prediction. This may be obvious to many interpreter writers, but not necessarily to hardware designers, and even the most prominent paper on interpreter performance characteristics [1] misses this point completely (see Section 7 for details).

*. Partially supported by Enterprise Ireland, International Collaboration Scheme
1. An earlier and much shorter version of this paper appears in the Euro-Par '01 proceedings [2].

As a consequence of the high number of indirect branches, the performance of efficient interpreters is highly dependent on the indirect branch prediction accuracy and the branch mispredict penalty; the magnitude of the performance impact probably surprises even interpreter writers: we see speedup factors of up to 4.77 between no predictor and a good predictor.

The main contribution of this paper is to quantify the effects that indirect branches have on interpreters on modern hardware, how this affects different implementation techniques, and how well existing and proposed indirect branch predictors work for interpreters.

Another important contribution of this paper is to counter the impression that readers of [1] may have that interpreters behave just like SPECint benchmarks.

1.1 Why efficient interpreters?

You may wonder why anyone should expend any effort on speeding up interpreters; why not just write a compiler to native code? The main reason is that a native code compiler requires significantly more effort in development and maintenance, in particular if it should support multiple platforms (see Section 1.2); the end result of taking the native code compiler route often is a language implementation that runs only on one platform, misses usability features like a decent debugger, takes noticeable time to compile, or is slow in tracking developments in the language or the environment.

Why not write a compiler to C? While this method addresses the ease-of-implementation and retargeting issues, it is not acceptable when interactivity, quick turnarounds, or non-C features (e.g., run-time code-generation or guaranteed tail-call optimization) are required.

Even if you are prepared to pay the costs of native-code compilation, you may still care for interpreter performance, because mixed-mode JIT compilers like HotSpot use an interpreter for the first executions of each piece of code; the interpreter produces profile data for determining where and how to expend the optimization effort. The faster the interpreter, the longer you can wait with the native-code compilation, resulting in compiling less code (and thus less compile time or better optimization) and better profile information (and thus optimization). The HotSpot system goes to great lengths to make the interpreter fast.

Some people argue that efficiency is not important in interpreters because they are slow anyway. However, this attitude can lead to an interpreter that is more than a factor of 1000 slower on many programs than native code produced by an optimizing compiler [1], whereas the slowdown for efficient interpreters is only a factor of 10 [3]. I.e., the difference between a slow and a fast interpreter is larger than the difference between a fast interpreter and native code.

Another argument is that interpreted programs spend much of their time in native-code libraries, so speeding up the interpretive engine will not provide a speedup to these programs. However, the amount of time spent in libraries is not necessarily known at the start of a project, and the prospect of slowdown by a factor > 1000 if the library does not cover everything and a significant amount of computation has to be performed using simple operations is daunting. Moreover, having an efficient interpreter increases the number of applications where the library dominates run-time. E.g., consider an application that would spend 99% of its time in libraries when compiled with an optimizing native code compiler: with an efficient interpreter (slowdown factor 10 for non-library code) it will spend 90% of the time in the library (for an overall slowdown of 1.1 over optimized native code), whereas with an inefficient interpreter (slowdown factor > 1000) it will spend less than 10% of its time in the library (for an overall slowdown of > 10).

Another question you may ask is: Why focus on *efficient* interpreters? Why not look at the most popular ones, such as Perl? Perl was already studied by Romer et al. [1], with the conclusion that it should be improved by software means. In this paper we are looking at interpreters where such means have already been applied. If the Perl community decides one day that interpreter efficiency is important (and it seems that they do for Perl6), the rewritten Perl interpreter will have performance characteristics similar to the efficient interpreters we are looking at in this paper, so this paper will also be useful for a version of Perl with a more efficient interpreter.

1.2 Native-code compilers

This section compares interpreters to native-code compilers quantitatively on speed and implementation effort:

The Ocaml system is a very good candidate for such a comparison, as it contains a good native-code compiler with many targets (currently 8 architectures) and also a fast interpreter. We look at Ocaml-3.06 in this section.

All compilers have 5995 lines of common code; the bytecode compiler has an additional 3012 lines; the native-code compiler has an additional 7429 lines of target-independent code and an additional 952–2526 lines for each target. In addition, the native code needs 1282 lines of target-independent run-time system code and 326–550 lines of target-specific run-time system code per target (and sometimes target/OS combination). Determining the size of the interpreter-specific run-time system is not easy (because it is not organized separately from the common (for all) run-time system, e.g., the garbage collector), but the bytecode interpreter itself has 1036 lines. The interpreter works for all targets supported by the native-code compiler and some additional ones, without target-specific code. Overall, even for only one target, the interpreter-based implementation is significantly smaller than the native-code implementation, and the difference grows with each additional target architecture.

Concerning execution speed, the Ocaml FAQ (http://caml.inria.fr/ocaml/speed.html) reports that the Ocaml native-code compiler is 2–15 (average: 6) times faster than the Ocaml interpreter.

Concerning compilation speed, on an 800MHz 21264 compiling a 4296 line file (`tk.ml`) takes 5.5s of CPU time with the bytecode compiler and 40.2s of CPU time with the native-code compiler, so the bytecode compiler is faster by about a factor of 7.

1.3 Overview

We first introduce interpreter implementation techniques (Section 2), then present our benchmarks, and their basic performance characteristics, in particular the high frequency of indirect branches (Section 3), then we introduce ways to make indirect branches fast (Section 4), and evaluate them on our benchmarks using a cycle-accurate simulator (Section 5). Finally we suggest some ways to speed up interpreters (Section 6). We discuss related work in Section 7.

2. Implementing efficient interpreters

This section discusses how efficient interpreters are implemented. We do not have a precise definition for *efficient interpreter*, but already the fuzzy concept "designed for good general-purpose performance" shows a direct path to specific implementation techniques.

If we want good *general-purpose* performance, we cannot assume that the interpreted program will spend large amounts of time in native-code libraries. Instead, we have to prepare for the worst case: interpreting a program performing large numbers of simple operations; on such programs interpreters are slowest relative to native code, because these programs require the most interpreter overhead per amount of useful work.

To avoid the overhead of parsing the source program repeatedly, efficient interpretive systems compile the program into an intermediate representation, which is then interpreted (this design also helps modularity).

To minimize the overhead of interpreting the intermediate representation, efficient interpretive systems use a flat, sequential layout of the operations (in contrast to, e.g., tree-based intermediate representations), similar to machine code; such intermediate representations are therefore called virtual machine (VM) codes.[2] Efficient interpreters usually use a VM interpreter, but not all VM interpreters are efficient.

Well-designed VMs are tailored for both easy compilation from the source language and fast interpretation on real machines. This makes it easy to write a fast compiler from the source language to the VM without losing too much performance due to interpretation overhead.

The interpretation of a VM instruction consists of accessing arguments of the instruction, performing the function of the instruction, and dispatching (fetching, decoding and starting) the next instruction. Dispatch is common to all VM interpreters and (as we will see in Section 5) can make up most of the run-time of an interpreter, so this paper focuses on dispatch.

Direct threaded code [4] is the most efficient method for dispatching the next VM instruction. Direct threaded code represents a VM instruction by the address of the routine that implements it (see Fig. 3). VM instruction dispatch consists of fetching the VM instruction and branching to the fetched addres, and incrementing the instruction pointer.[3]

Unfortunately, direct threading cannot be implemented in ANSI C and other languages that do not have first-class labels and do not guarantee tail-call optimization.

Fortunately, there is a widely-available language with first-class labels: GNU C (version 2.x); so direct threading can be implemented portably (see Fig. 1). If portability to machines without `gcc` is a concern, it is easy to switch between direct threading and ANSI C conforming methods by using macros and conditional compilation.

2. Romer et al. [1] use *virtual machine* in a wider sense, basically encompassing all interpreters, including string interpreters.
3. Direct threaded code is the most efficient method in actual use, because all methods for VM instruction dispatch use an indirect branch or call (with the branch address providing either the same or less context to be used in branch prediction), and direct threaded code uses the least additional overhead. It is an interesting question whether using a tree-of-conditional branches for dispatch would result in faster dispatch than using an indirect branch; we doubt it, because VM instruction dispatch has many possible targets, without strong bias, so such a dispatch would always run through several conditional branches, which costs time even if they are all correctly predicted; and there is no guarantee that there will be fewer mispredictions than with indirect branches.

```
typedef void *Inst;

void engine()
{
  static Inst program[] = { &&add /* ... */ };
  Inst *ip = program;
  int *sp;

  goto *ip++; /* dispatch first VM inst. */

 add:
  sp[1]=sp[0]+sp[1];
  sp++;
  goto *ip++; /* dispatch next VM inst. */
}
```

Figure 1: Direct threaded code using GNU C's "labels as values"

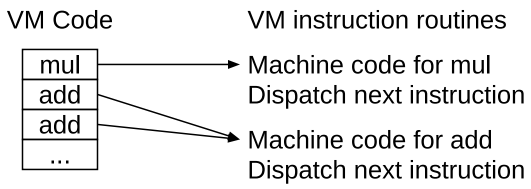
```
typedef enum {
  add /* ... */
} Inst;

void engine()
{
  static Inst program[] = { add /* ... */ };

  Inst *ip = program;
  int *sp;

  for (;;)
    switch (*ip++) {
    case add:
      sp[1]=sp[0]+sp[1];
      sp++;
      break;
    /* ... */
    }
}
```

Figure 2: Instruction dispatch using `switch`



```
lw   $2,0($4) #get next inst., $4=inst.ptr.
addu $4,$4,4  #advance instruction pointer
j    $2       #execute next instruction
#nop          #branch delay slot
```

Figure 3: Direct threaded code and its dispatch sequence in MIPS assembly

```
$L2: #for (;;)
 lw    $3,0($6) #$6=instruction pointer
 #nop
 sltu  $2,$8,$3 #check upper bound
 bne   $2,$0,$L2
 addu  $6,$6,4  #branch delay slot
 sll   $2,$3,2  #multiply by 4
 addu  $2,$2,$7 #add switch table base ($L13)
 lw    $2,0($2)
 #nop
 j     $2
 #nop
 ...

$L13: #switch target table
 .word   $L12
 ...

$L12: #add:
 ...
 j   $L2
 #nop
```

Figure 4: Switch dispatch in assembly

Implementors who restrict themselves to ANSI C usually use the giant switch approach (Fig. 2): VM instructions are represented by arbitrary integer tokens, and the switch uses the token to select the right routine.

Figure 3 and 4 show MIPS assembly code for the two techniques. Both methods perform one indirect branch per executed VM instruction, so when interpreting programs with mostly simple VM instructions, the proportion of executed indirect branches is relatively high (see Section 3), and their impact on run-time is even higher (see Section 5.2.2).

The execution time penalty of the switch method over direct threading is caused by a range check, by a table lookup, by the branch to the dispatch routine generated by most compilers, and, as we will see in Section 5.2.4, on some processors by a higher indirect branch mispredict rate.

3. Benchmarks

This section describes the interpreters we have benchmarked and gives some numbers on the behaviour (in particular the frequency of indirect branches) of these interpreters. It also discusses why Perl and Xlisp are not representative of efficient interpreters.

The interpreters we benchmarked are[4]:

Gforth 0.4.9-19990617, a Forth system [5]; it uses a virtual stack machine. We compiled it for indirect threaded code[5] [6] (i.e., there is one additional load instruction between the instruction load and the indirect branch). The workloads we use are *prims2x*, a compiler for a little language, and *bench-gc*, a conservative garbage collector.

Ocaml 2.04, the Objective CAML "bytecode"[6] interpreter, which also implements a virtual stack machine [7]; the VM works on tagged data (with some associated overhead), but does not do run-time type-checking; Ocaml uses tagged data to support polymorphism and garbage collection. We ran both, a version using direct-threaded code, and a switch-based version (Ocaml is prepared for that, we just had to change a configuration option). The workloads we use are *ocamllex*, a scanner generator, and *ocamlc*, the Ocaml compiler.

Scheme48 0.53; this interpreter is switch-based, uses a virtual stack machine, uses tagged data and performs run-time type checking [8]. We use the Scheme48 compiler as workload (the first 10^8 instructions of building scheme48.image).

Yap 4.3.2, a Prolog system, uses an interpreter based on the WAM, a virtual register machine; it uses tagged data and performs run-time type-checking [9]. As with ocaml, we ran a version using direct threaded code and a version using switch dispatch. The workloads we use are *boyer*, part of a theorem prover, and *chat_parser*, a parser for English.

Perl the SPEC95 benchmark. This interpreter uses a linked-list intermediate representation and interprets that. We use the SPEC95 train/jumble benchmark as workload.

Xlisp the SPEC95 benchmark *li*. Xlisp interprets S-expressions (a list/tree-based intermediate representation of Lisp data). We use the SPEC95 ref/boyer[7] program as workload.

4. You can find these benchmarks through http://www.complang.tuwien.ac.at/anton/interpreter-arch.
5. We could not compile Gforth for direct threaded code on SimpleScalar.
6. Actually the instructions take 32 bits.
7. The Lisp Boyer is related to the Prolog Boyer we use with Yap, but the results are probably not comparable.

interpreter	workload	inst. exec.	loads	stores	branches	indirect-branches		inst./ind.
gforth	benchgc	64396699	40.7%	10.5%	14.5%	13.0%	8380089	7.6
gforth	prims2x	94962783	39.4%	8.8%	18.4%	10.3%	9841564	9.6
ocaml	ocamlc	66439963	26.4%	10.2%	17.5%	6.3%	4204772	15.8
ocaml (switch)	ocamlc	91550037	21.8%	6.1%	21.5%	4.5%	4204772	21.7
ocaml	ocamllex	69738725	29.5%	10.7%	19.8%	11.3%	7918321	8.8
ocaml (switch)	ocamllex	122558868	22.2%	5.1%	24.3%	6.4%	7918321	15.4
scheme48	build	100000003	27.9%	12.1%	20.0%	3.2%	3275171	30.5
yap	boyer	68153580	32.9%	11.7%	19.7%	5.4%	3681492	18.5
yap (switch)	boyer	97326209	24.8%	8.2%	24.2%	3.7%	3681492	26.4
yap	chat	15510382	33.4%	14.5%	17.1%	5.5%	864703	17.9
yap (switch)	chat	22381662	25.6%	10.0%	22.4%	3.8%	864703	25.8
perl	jumble	2391904893	25.8%	17.8%	19.3%	0.7%	17090181	140.0
xlisp	boyer	173988551	26.1%	16.8%	22.7%	1.1%	1858367	93.6

Figure 5: Instruction class distribution for our benchmarks

We use Gforth, Ocaml, Scheme48, and Yap as examples of efficient interpreters[8]. We believe that these interpreters are efficient because the papers written about them [5, 7, 8, 9] indicate that their authors cared about performance; for Scheme48 performance seems to be less important than for the other efficient interpreters, but it is still a fast system and shares the performance characteristics discussed in this paper with the other efficient interpreters. Another indication of the efficiency of Gforth and the Ocaml bytecode interpreter is that they are the fastest interpreters in many of the benchmarks in the "Computer Language Shootout" (http://www.bagley.org/~doug/shootout/).

We include Perl and Xlisp for comparison with related work [10], and because they are in the Spec95 benchmark suite (they are also the most indirect-branch-intensive benchmarks in the Spec95 suite). We selected workloads of non-trivial size and used two workloads for most interpreters in order to exercise significant portions of the interpreters and to avoid overemphasizing some non-representative aspect of the performance.

We compiled the interpreters for the SimpleScalar 2.0 microprocessor simulator [11], which simulates a MIPS-like architecture. Some basic data on the behaviour of these benchmarks is listed in Fig. 5.

The column *inst. exec.* contains the number of executed (and commited) instructions in the simulation. The next columns specify the proportion of loads, stores, branches (all control transfers), and indirect branches (without returns); then you see the absolute numbers of indirect branches, and finally the executed instructions per indirect branch (i.e., the inverted indirect branch proportion column). We excluded returns from the indirect branch numbers, because they are not used for interpreter dispatch, and because they can be easily and accurately predicted with a return-address stack.

8. You may wonder about the absence of JavaVM interpreters. The reason is that we don't know an efficient JavaVM interpreter that runs on SimpleScalar-2.0.

The aspect of these data that is important for this paper is the extremely high proportion of indirect branches in the efficient interpreters: 3.2%–13%[9]; in comparison, the benchmarks used in indirect branch prediction research [10] perform at most 2.1% indirect branches.

The reason for all these indirect branches is that every VM instruction performs at least one indirect branch, whether it uses threaded code or switch dispatch. Most VM instructions are quite simple (e.g., add the top two numbers on the stack), and can be implemented in a few native instructions plus dispatch code. Complex VM instructions occur relatively rarely, and the setup for them typically takes many simple VM instructions; moreover, complex operations are sometimes split into several VM instructions, because this allows optimizing special cases in the compiler and/or because it simplifies the interpreter (similar to the CISC/RISC transition).

The higher number of native instructions per VM instruction for Yap and Scheme48 can be explained by the dynamic typechecking required by Scheme and Prolog; moreover, Scheme48 uses switch dispatch and Yap's VM is a register machine, which often requires more overhead in instruction decoding and operand access than a stack machine (however, a sophisticated compiler may generate fewer of these instructions). Also, Scheme48 sacrificed some efficiency for other goals.

Another issue that can be seen nicely is the effect of threaded code vs. switch dispatch on the instruction count; the number of indirect branches is exactly the same in both cases, but the total number of instructions is higher for interpreters using switch dispatch (by a factor of 1.76 for ocamllex). For Ocaml switch dispatch costs 5.9–6.6 instructions more than threaded code, for Yap the difference is 7.9 instructions. Section 5.2.4 shows how these differences affect execution time.

It is also clearly visible that Perl and Xlisp do not share the indirect branch characteristics of the efficient interpreters.

4. Fast indirect branches

Like all branch instructions, indirect branches are potentially slow in current CPUs, because they require flushing the pipeline if mispredicted. Unfortunately, many CPUs either do not predict indirect branches at all, or do so poorly. In this paper we look at several architectural and microarchitectural ways to predict indirect branches and how well they work with various VM interpreters.

Unless special measures are taken, conditional branches and indirect branches are relatively expensive operations in modern, pipelined processors. The reason is that they typically only take effect after they have reached the execute stage (stage 10 in the Pentium III, stage 17 in the Pentium 4) of the pipeline, but their result affects the instruction fetch stage (stage 1); i.e., after such a branch the newly fetched instructions have to proceed through the pipeline for many cycles before reaching the execute stage. The resulting delay is known nowadays as the misprediction penalty. In contrast, the result of an ordinary ALU operation is available at the end of the execute stage and usually needed by later instructions at the start of the execute stage, resulting in a latency of one cycle between ALU instructions and later instructions.

9. Note that Gforth requires one load less per indirect branch if it uses direct-threaded instead of indirect-threaded code, resulting in an even higher proportion of indirect branches.

Of course, this problem has been attacked with a variety of architectural and microarchitectural methods. However, most efforts focussed on conditional branches and on procedure returns, with indirect branches being neglected; even today, several CPUs are sold without any way to reduce the cost of indirect branches (e.g., AMD K6-2, MIPS R10000, UltraSPARC II).

4.1 Indirect Branch Prediction

One way to avoid the mispredict penalty is to correctly predict the target of the branch and execute instructions there speculatively. Of course, if the prediction is incorrect, the speculatively executed instructions have to be canceled, and the CPU incurs the misprediction penalty.

4.1.1 Profile-Guided Prediction

The Alpha architecture has indirect branch target hints: a field in the JMP instruction specifies 16 bits of the predicted branch target (this should be enough to predict an instruction in the I-cache). This can be used by a compiler with profile feedback to predict indirect branches. Unfortunately, the compilers we know don't support this in a way that helps interpreters: gcc-2.95.2's "-fprofile-arcs does not support computed gotos"; the Compaq C compiler supports only switch dispatch and produces only one indirect branch, which limits the prediction accuracy to about 10% (see Section 5). Another disadvantage of this prediction method is that the branch has to be decoded first, meaning that even with a correct prediction there is a branch execution penalty (the cycles before the decode stage).

4.1.2 Dynamic Prediction

Dynamic branch prediction is a microarchitectural mechanism and works without architectural, compiler or software support (but software support can help, see Section 6.3).

The simplest dynamic indirect branch prediction mechanism is the branch target buffer (BTB), which caches the most recent target address of the branch; it uses the address of the branch instruction as the key for the lookup.

An improvement on the normal BTB is the BTB with 2-bit counters, which replaces a BTB entry only when it mispredicts twice[10]; this halves the mispredictions if a branch usually jumps to just one target, but with a few exceptions. This kind of BTB was introduced in 1993 with the Pentium, and is still the most sophisticated indirect branch prediction method available in commercially available CPUs.

The best currently-known indirect branch prediction mechanisms are two-level predictors that combine a global history pattern of n indirect branch targets with the branch address and use that for looking up the target address in a table; these predictors were proposed by Driesen and Hölzle [10], and they studied many variations of these predictors; one issue they did not study (probably because they used a trace-driven instead of a an execution-driven simulator) is the problem that the real target of the branch is only avail-

10. Actually one bit is sufficient, if the BTB is used only for indirect branch prediction. This name is used because the corresponding structure in conditional-branch predictors uses two-bit saturating counters. And, e.g., the Pentium has two bits per entry, because the entries are also used for conditional branch prediction.

able long after the prediction and cannot be used immediately for updating the history pattern and the tables (see Section 5).

We discuss the effects of various prediction mechanisms on interpreters in Section 5.

4.2 Mispredict Penalty

Another way to reduce the cost of branches is to reduce the misprediction penalty instead of or in addition to reducing the number of mispredictions.

One approach is to use delayed branches. Delayed branches fell out of favour in architectures for desktop and server machines because for maximum benefit they would have an implementation-dependent number of delay slots, which would break binary compatibility. However, in various CPUs for embedded systems delayed branches are used aggressively, and they can be used very well when implementing interpreters [3].

A more flexible (less implementation-dependent) approach than delayed branches are split branches, where a prepare-to-branch instruction specifies the branch target, and the point where the branch actually takes effect is specified explicitly. In the Power/PPC, HP PlayDoh, and IA64 architectures the prepare-to-branch instruction specifies a branch register to contain the target address, but other approaches are possible [12].

As an example, let's look at the Power/PPC architecture: The prepare-to-branch instruction is called `mtctr`, and the actual control transfer instructions is `bctr`.[11] In the first Power implementation (and many successors) the latency between these instructions is five cycles (used for getting the address into the branch unit, and to start fetching and decoding the target), but the actual control transfer then takes place immediately.

However, with both delayed and split branches, the processor has to fetch, decode, and execute the load instruction that loads the next VM instruction address before the jump to that address can take effect. This would mean that executing a VM instruction takes at least as long as it takes a load to get from the instruction fetch stage to the result stage of the processor.

This problem can be reduced by moving the load into the previous VM instruction, essentially software pipelining the interpreter [3]. However, we have to rename the result register when we move the load across the previous jump. The simple way to do this is to use a move instruction[12] (the prepare-to-jump instructions of PowerPC, PlayDoh, and IA-64 are a kind of move instruction, so there is no extra instruction necessary there); however, this move instruction also has to be fetched, decoded, and executed before the jump can take effect, determining the minimum VM instruction execution time: fetch, decode, and execute the move, then execute the indirect jump.

Another problem with these approaches is suboptimal compiler support: E.g., gcc-2.95 does not schedule the prepare-to-branch instruction as early as one would like.

11. There is also the pair `mtlr`/`blr`, which uses another branch register, but it is intended for return addresses only, and the PPC 7450 predicts the `blr` target using a return stack.
12. The other way is to duplicate the whole interpreter to perform modulo variable renaming, but this is quite complex, especially for threaded code.

5. Evaluation

This section shows how the mispredict penalty for indirect branches affects interpreters and how various branch prediction schemes help.

5.1 Simulation setup

We added several branch prediction methods to the SimpleScalar-2.0 simulator, and ran the interpreters with the following prediction methods:

No prediction The processor just has to wait until the branch resolves.

Profile-Guided Predicts that an indirect branch always jumps to the address that was most frequent in a training run. We generally use a different interpreted program for the training run (except for Scheme48, but it still gets a very low prediction accuracy).

BTB Predicts that a branch always jumps to the target it jumped to last time (unlimited table size).

BTB2 A BTB that only replaces the prediction on the second miss (unlimited table size).

2-Level Two-Level predictors that use the branch address and a history register containing the last 1–8 target addresses as a key into a lookup table of predicted targets and two-bit-counters [10]. We use all the bits of the address and a table of unlimited size. You can see BTB2 as 2-Level predictor with 0 target addresses in the history register. We cannot use this predictor in the cycle-accurate simulator, because the processor does not know the real target address until the instruction has commited. Therefore we use two variants of this predictor:

2-Level-speculative Like 2-level, but the history register contains the last n predicted targets. The history register is updated speculatively with the predicted target when the instruction fetch from the predicted target occurs. If a misprediction is detected at any time, the history register stays the same (no roll-back is performed).

2-Level-commited Like 2-level, but the history register contains the last n commited addresses (note that the commit can happen after another indirect branch has already been predicted).

SimpleScalar can vary the mispredict penalty. This allows us to get some idea of the effects of various hardware techniques for reducing the misprediction penalty. SimpleScalar simulates an out-of-order processor with a short pipeline: Instruction Fetch, Decode/Dispatch, Execute, Writeback, Commit. An indirect branch has to wait in dispatch until the target address becomes available; then it proceeds through Execute and Writeback; if the branch was mispredicted, it fetches instructions from the correct address in the cycle after the instruction loading the address performs Writeback, if we set the mispredict penalty to a minimum; this is usually called a mispredict penalty of four cycles (SimpleScalar calls it 1 cycle, but we follow the more traditional usage throughout the paper); note that out-of-order execution can raise the real mispredict penalty even higher,

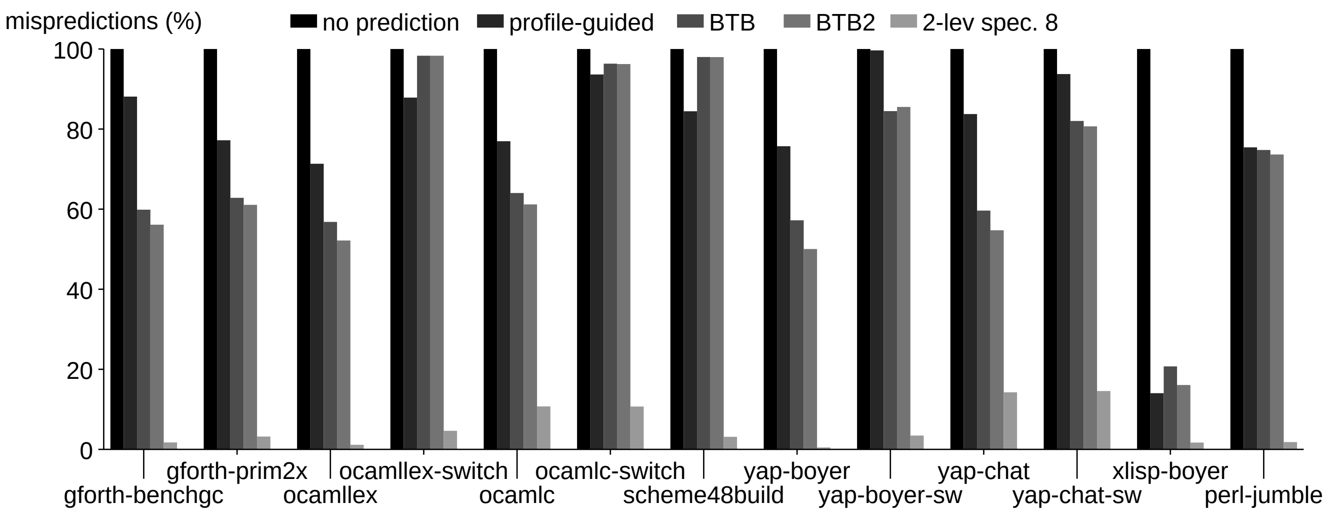


Figure 6: Mispredict rate of various predictors on interpreters

because reaching the writeback stage is delayed if the branch has to wait for results of other instructions (i.e., the load providing the target address).

If we set the mispredict penalty higher, a corresponding number of cycles will elapse between the time that the branch is in Execute and the time when the correct instructions are fetched; you can see this as emulating a processor with more instruction fetch and decode stages.

The unlimited table size is unrealistic for hardware implementation, but we think that the results are more interesting than looking at one particular realistic configuration, because that configuration might produce confusing artifacts (e.g., from conflict misses) that would not occur for another configuration, or after small code changes. Also, at least for two-level predictors the unlimited model is quite close to well-tuned realistic models with large table sizes [10].

Apart from the indirect branch predictor and the mispredict penalties we mostly used the default processor configuration of SimpleScalar-2.0: a superscalar processor capable of issuing four instructions per cycle, with the usual additional restrictions (e.g., only two L1 cache ports); the load latency from L1 cache is two cycles. The conditional branch predictor we configured is a little better than the default: a combined predictor consisting of a bimodal and a two-level adaptive predictor, with 4096 entries in each table and a history length of 12 for the two-level predictor. We also increased the size of the return-address stack to 64.

5.2 Results

5.2.1 Prediction accuracy

Figure 6 shows the mispredict rates of the interpreters. We see four different patterns in indirect branch prediction accuracy:

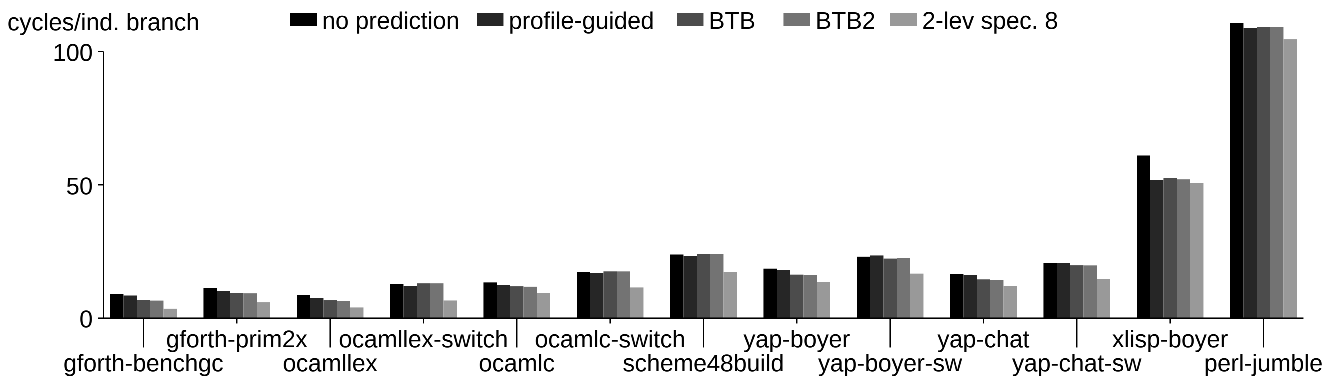


Figure 7: Execution time of various interpreters with various branch prediction scheme with a mispredict penalty of 4 cycles ("1 cycle" in SimpleScalar terminology)

Threaded code Profile-guided prediction produces 71%–88% mispredictions, BTB 57%–63%, BTB2 50%–61%.

Switch-based code Profile-guided prediction produces 84%–100% mispredictions. BTB and BTB2 produce 98% mispredictions for ocaml and scheme48, and 81%–86% for Yap.

li Profile-guided prediction, BTB, and BTB2 perform quite well (14%–21% mispredictions). The reason for this result is that most dynamic indirect jumps in li perform dynamic type resolution, not interpreter dispatch, and apparently the boyer program deals mainly with one specific data type.

perl Profile-guided prediction, BTB, and BTB2 produce about 75% mispredictions.

The prediction accuracy of two-level predictors is relatively good (typically less than 10% mispredictions) for all interpreters.

The reason for the difference between the accuracies for threaded code and switch-based code is that threaded code uses a separate dispatch routine for each VM instruction, which allows profile-guided prediction and the BTB to predict a different target for each VM instruction. I.e., with separate dispatch routines the predictor has more context to use for predictions (a kind of one-deep history).

In contrast with a common dispatch jump, these predictors just have to predict the same thing irrespective of where they are coming from, leading to worse performance. The profile-guided predictor just predicts the most frequently used VM instruction. The BTB predicts the last executed instruction; apparently this prediction is more accurate for the register-based Yap than for the stack-based Ocaml and Scheme48 interpreters.

5.2.2 Effect on execution time

Figure 7 shows the effect of the different prediction accuracies on the execution times, assuming a short misprediction penalty. To get a better view of the details, Fig. 8 shows

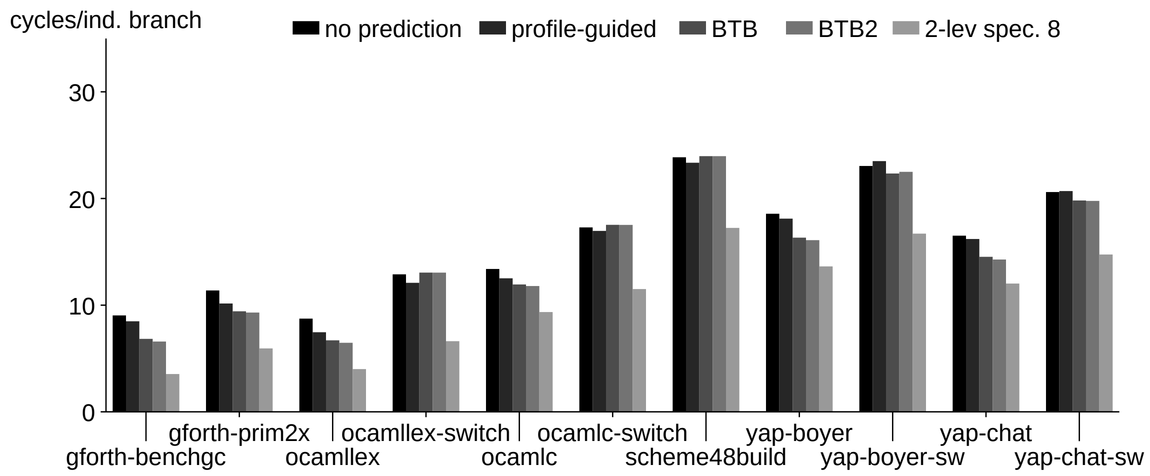


Figure 8: Execution time of *efficient* interpreters with various branch prediction scheme with a mispredict penalty of 4 cycles ("1 cycle" in SimpleScalar terminology)

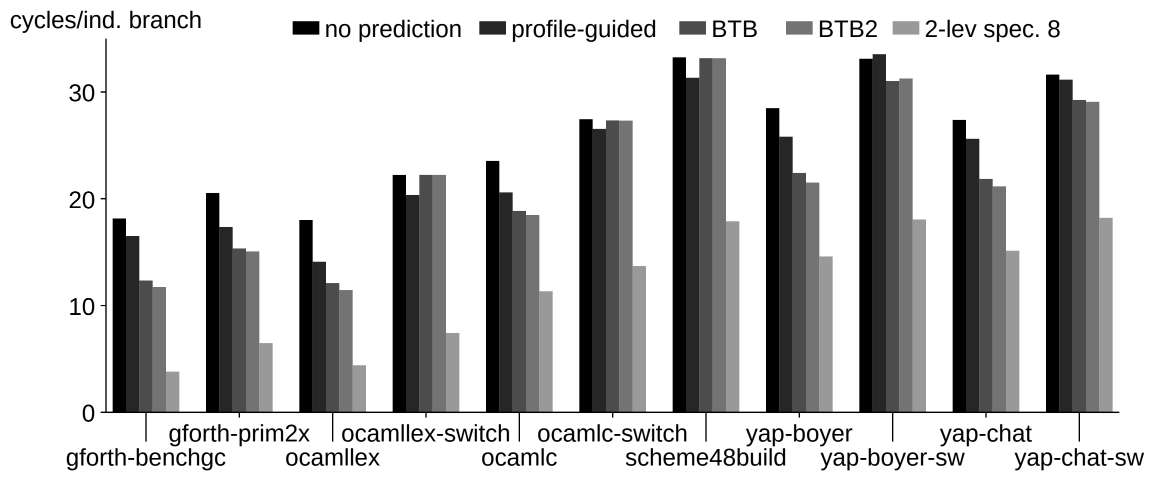


Figure 9: Execution time of *efficient* interpreters with various branch prediction scheme with a mispredict penalty of 13 cycles ("10 cycles" in SimpleScalar terminology)

the same data (without li and perl) at a different scale. In the efficient interpreters nearly all indirect branches perform a VM instruction dispatch, so you can assume that the cycles per indirect branch also represent cycles per VM instruction.

For the most part, the difference in prediction accuracy between no prediction and "2-level spec. 8" results in an execution time difference mostly between 4 cycles (ocamlc) and 6.6 cycles (scheme48build), with an outlier at 10.4 cycles (li-boyer). Note that these differences can be larger than the pure branch prediction penalty because the indirect jump can depend on other instructions that have to be resolved first.

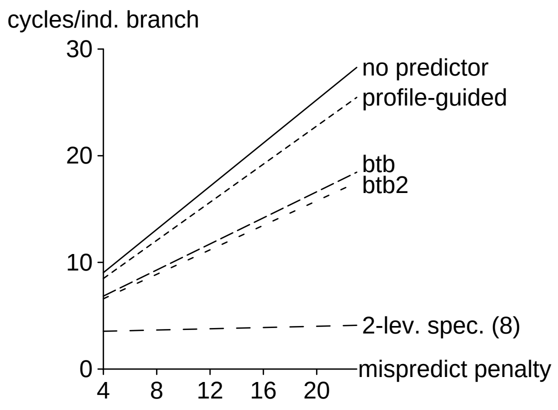


Figure 10: Execution time for gforth-benchgc for varying penalties and indirect branch predictors. The other benchmarks show similar, linear behaviour.

The difference between the efficient interpreters and the others is that for the efficient interpreters this difference constitutes a much larger percentage of the execution time. E.g., gforth-benchgc with “2-level spec. 8” takes 3.5 cycles per indirect branch, whereas with no branch prediction it takes 9 cycles, i.e, there is an execution time factor of 2.55 to be gained from branch prediction, or viewed differently, without indirect branch prediction this benchmark spends 61% of its time on indirect branches. By contrast, the factor between no prediction and “2-level spec. 8” is much less prominent for li-boyer (1.20) and perl-jumble (1.06).

With longer, more realistic misprediction penalties (Fig. 9) the effect of mispredictions on run-time is even more dramatic. E.g., with 13 cycles misprediction penalty the factor between no prediction and “2-level spec. 8” is 4.77 for gforth-benchgc (or 79% of the time in indirect branches).

5.2.3 Varying the mispredict penalty

Figure 10 shows how the run-time of one of the benchmarks varies with the mispredict penalty: it rises nearly linearly with the mispredict penalty. The other benchmarks also have linear behaviour.

This surprised us: We originally expected that, given a string of correct predictions and sufficient fetch/decode bandwidth, interpreter dispatch would get ahead of the rest of the VM instruction execution because dispatch has fewer dependences (just one instruction pointer update per VM instruction); then the overlap between executing old VM instructions and the misprediction should flatten the curves for low mispredict penalties in Fig. 10. However, such overlap does not happen in significant amounts (we looked at some traces, and also experimented with configurations with more resources to confirm this). The reason for this is that the loads in the dispatch depend on stores in the rest of the VM instruction execution (in particular, stack spills in stack VMs and result register writes in register VMs). We will probably see somewhat different results on a processor that can speculatively execute loads even before stores with yet-unknown addresses.

predictor	cycles
none	3.9–4.5
profile-guided	4.4–5.4
BTB	5.3–6.3
BTB2	5.5–6.6
2-level spec. (8)	2.2–3.1

Figure 11: Speed difference between threaded code and switch dispatch for various branch predictors

One nice thing about this linear behaviour is that it makes analysing and predicting the behaviour easy: halving the number of mispredictions (of all kinds of branches) has the same effect as halving the mispredict penalty (including the latencies of unfinished instructions the indirect branch depends on). Starting with current hardware (i.e., BTB-style indirect branch predictors), it is probably much easier to speed up efficient interpreters by adding a 2-level indirect branch predictor than by shortening the mispredict penalty by an equivalent amount.

5.2.4 Threaded code vs. switch dispatch

The run-time difference between threaded code and switch dispatch consists of differences coming from the different mispredict rates, and from differences due to the additional executed instructions. Figure 11 shows the overall difference between threaded code and switch dispatch on SimpleScalar with 4 cycles mispredict penalty (1 cycle in SimpleScalar terminology).

The difference is larger for profile-guided, BTB, and BTB2 than without predictor because switch dispatch has significantly more mispredictions with these predictors. The difference is smaller for the two-level predictor than without predictor, because switch dispatch and threaded code have similar mispredict rates with this predictor, and the correct predictions allow hiding the latencies of the switch code by overlapping it with later code.

Given the short execution time of the threaded-code VM instructions, using switch dispatch is slower by up to a factor of 2.02 (ocamllex with BTB2 and 4 cycles mispredict penalty).

5.2.5 Inefficient interpreters

The behaviour of Xlisp and Perl is very different from the other interpreters. Xlisp has a low misprediction rate for all predictors. We examined the code and found that most dynamically executed indirect branches do not choose the next operation to execute, but are switches over the type tags on objects. Most objects are the same type, so the switches are quite predictable. The misprediction rates for Perl are more in line with other interpreters, but figure Fig. 7 shows that improving prediction accuracy has little effect on Perl. Non-return indirect branches are simply too rare (only 0.7%) to have much effect.

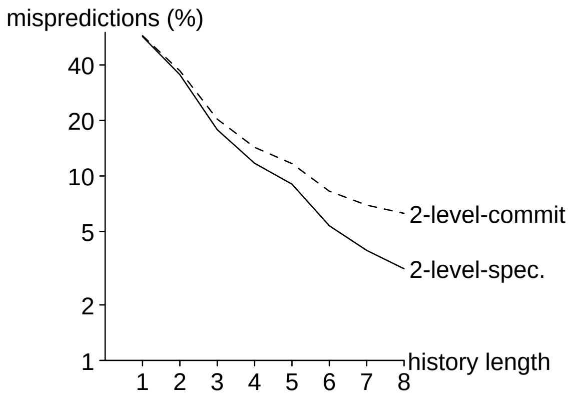


Figure 12: Two-level indirect branch predictor accuracy for Scheme48 build

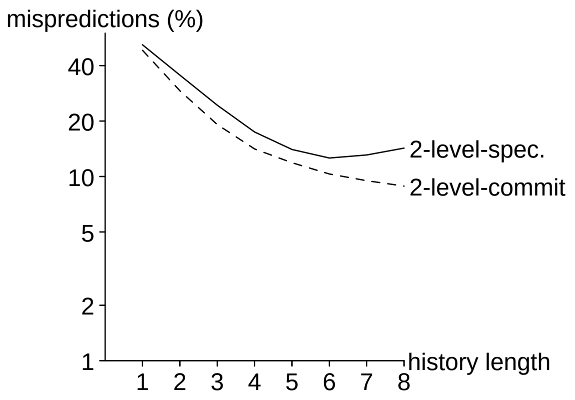


Figure 13: Two-level indirect branch predictor accuracy for Yap chat

5.2.6 Two-level predictors

In this section we diverge to take a closer look at our variations of two-level indirect branch predictors. Note that this section is not as deep as some might wish because indirect branch predictors are not the main topic of the paper.

We have measured 2-level-speculative and 2-level-commited predictors with history lengths of up to eight. There are significant variations between the benchmarks; we think that these are due to differences in the interpreted programs rather than in the interpreters. Figure 12 and 13 show two examples. In any case, both predictors usually work quite well given a sufficient history length.

Longer histories are usually, but not always better; our explanation for this is that the predictors train themselves to inner loops or frequently-executed procedures of the interpreted programs; if the trip counts of the loops are small, longer histories have too much overhead. For some benchmarks the speculative predictor is better, for others the commit-based predictor. This suggests that 2-level-speculative and 2-level-commited might be useful as components of a hybrid predictor [13, 10].

5.2.7 Real-world results

This section supports the validity of the simulation results by presenting data from a real machine. We produced the data by using the performance monitoring counters of the Athlon. The data are not directly comparable, but the results support some of our conclusions from the simulation results. The differences are: the Athlon has a different architecture than SimpleScalar and counts different events (in particular, we have to count *taken branches* as the closest available approximation to indirect branches); and we used a newer version of Gforth, because it supports an optimization that allows varying some parameters. We use the gforth-benchgc benchmark because that benchmark performs relatively few direct branches, so counting taken branches as approximation to the number of indirect branches should work best in this benchmark.

We measured that 55% of the taken branches are mispredicted in gforth-benchgc (without optimization), which is not far from the indirect branch mispredict rate of 60% that we see for a BTB on SimpleScalar (the Athlon uses a BTB).

When activating the superinstruction optimization (see Section 6.3) that reduces the number of executed real machine instructions by a factor of 1.22, the number of taken branches by a factor of 1.49 and the number of taken-branch mispredictions by 2.28, we see that the execution time reduction (factor 1.74) is higher than the reduction in executed instructions and taken branches. This indicates that the number of mispredictions is more important for the execution time of this benchmark than the number of executed instructions.

Gforth-benchgc (without superinstructions) performs a misprediction every 20 cycles. The Athlon has a branch mispredict penalty of 10 cycles, so it is easy to see that this benchmark spends half of its time in mispredictions (and maybe some more of its time waiting for the target address).

6. Advice

6.1 Hardware designers

The most useful hardware support feature for efficient interpreters is good indirect branch prediction. Two-level predictors work well for interpreters as well as for other applications involving indirect branches or indirect calls (e.g., object-oriented programs). For the interpreters we measured, a two-level predictor achieved a speedup over BTB2 of up to a factor of 1.97 for a mispredict penalty of 4 cycles and up to a factor of 3.08 for a mispredict penalty of 13 cycles.

Even just having a BTB helps threaded code quite a bit, and with appropriate changes in the interpreter [14] their prediction accuracy can be very good.

6.2 C compiler writers

C compiler writers can improve the performance of switch-based interpreters (and possibly other software) by replacing the single, shared switch dispatch code fragment with a copy of the dispatch fragment for each VM instruction. This can be achieved by eliminating the unconditional branches back to the switch code (tail duplication). This optimization would increase the prediction accuracy to the same level as we see with threaded code. We estimate that this optimization can provide a speedup factor of up to 1.5 on a processor with a BTB2 predictor.

Another way a C compiler could help interpreter performance is by supporting GNU C's labels-as-values extension, which allows to implement threaded code. Threaded code can provide a speedup factor of 2.02 over switch dispatch with a shared dispatch fragment (see Section 5.2.4).

Alternatively, guaranteeing tail-call optimization would be another way to support threaded code [15].

Supporting indirect branch target hints on architectures that support them, preferably by profile feedback, would also help a little (up to a factor of 1.17).

6.3 Interpreter writers

A number of techniques for writing efficient interpreters have been described in the literature (see Section 7). This section focusses on techniques for reducing the time spent in indirect branches, and their effect.

Given the slow adoption of BTBs in the past, we can expect at least a decade to pass before more sophisticated indirect branch prediction hardware will be available on most CPUs, so interpreter writers will have to cater for CPUs with BTBs and possibly without indirect branch prediction in the near future.

An interpreter writer should support threaded code (see Section 2), for a speedup of up to a factor of 2.02 over a switch-based interpreter; to maintain maximum portability, a switch-based ANSI C version of the interpreter should be available through conditional compilation.

Following the present work, we looked on how to improve interpreters for improved BTB prediction accuracy [14]. We found that replicating the code (and thus the indirect branch) of VM instructions and using the different replicas in different places can reduce the mispredictions significantly, similar to the effect we have seen here from using a separate dispatch branch per VM instruction. In particular, if we have a separate replica for each instance of a VM instruction in the program, the only mispredictions we have are for VM-level indirect branches and mispredictions from BTB conflicts, resulting in a prediction accuracy of over 90% and speedups by factors of up to 2.39 on a Celeron-800 (and 2.41 on an Athlon-1200) [14].

Combining common sequences of VM instructions into VM superinstructions [16, 17] also helps by introducing more static indirect branches, and also by reducing the number of dynamically executed indirect branches (the latter effect also helps processors without BTBs). The combination of replication and superinstructions produces speedups by up to a factor of 3.17 on a Celeron-800 [14].

There are two ways of implementing these techniques: Statically compiling a given set of replicas and/or superinstructions into the interpreter; or dynamically copying code from the static copy of the interpreter. Since the dynamic methods can adapt to the program at hand, they produce better BTB prediction accuracies and (on processors like Celeron or Athlon) better speedups. They are also somewhat easier to implement, but require some target-specific code (mainly for flushing the I-cache), and more memory. If these problems are acceptable for your application, we recommend using dynamic superinstructions with dynamic replication [14].

7. Related Work

We already discussed the study by Romer et al. [1] in Section 1. The most important difference from our work is that the interpreters they studied were not representative of efficient interpreters; only Java uses a VM[13], but not a very efficient one: it needs on average 16 native instructions for fetch/decode (in contrast to 3 native instructions for threaded-code dispatch) and 18–27 (170 on Hanoi) native instructions for the rest per JavaVM instruction, although the VM is similar to Gforth's and Ocaml's VM (which need 7.6–15.8 instructions/VM instruction *total*); on the DES benchmark Java is 119 times slower than C. Branch mispredictions took less than 10% of the cycles on a 21064 for all benchmarks on all interpreters they measured. In contrast, this paper studies interpreters written for high performance where further performance improvements would require significant increases in complexity. Branch misprediction is a major component of run-time for the interpreters we studied.

The basis for our two-level indirect branch predictors is Driesen and Hölzle's work [10]. They start out with the 2-level predictor, and then explore what happens when they add more and more restrictions to make the predictor implementable in hardware. A major difference between our work and theirs is that we use efficient interpreters as benchmarks, which have much higher indirect branch frequencies (up to 13% vs. up to 2.1%) than any of their benchmarks (mostly object-oriented programs, particularly those with higher indirect branch frequencies), and typically also more potential targets per indirect branch. We limited ourselves to using only one of their predictors (exploring predictors was not our primary goal), but we explored one issue that they did not study: the fact that the correct result is only available long after the prediction and how to deal with it.

They have also presented an improvement, multi-stage cascaded prediction, which reduces the size of the required tables [18, 19]. Another promising approach takes advantage of predictors from data compression techniques to predict the target of indirect branches [20, 21]. The same authors also demonstrate how indirect branches can be more accurately predicted by classifying them into groups [22]. An earlier approach to two level indirect branch prediction was to use the recent history of conditional branch outcomes to predict indirect branches [23]. Although the results are better than for a BTB, more recent schemes are more accurate.

The accuracy of profile-guided conditional branch prediction can be improved by code replication [24, 25]. Similarly, replicating the dispatch code improves the performance of profile-guided indirect branch prediction (as well as BTB performance) by providing more context.

A very large body of work exists on conditional branch prediction. The most important is the two-level adaptive predictor, invented independently about the same time by Yeh and Patt [26] and Pan et al. [27]. Another important discovery was the link between branch

13. MIPSI emulates a real machine (MIPS) directly. Performance can be improved by translation into a VM, but in this application the tradeoff between complexity and performance is less favourable for the VM approach than in a programming language interpreter.

prediction and predictors used for data compression [28], which established a theoretical basis for branch prediction. Uht et al. [29] give a good introduction to the many existing direct branch prediction schemes.

Much of the knowledge about interpreters is folklore. The discussions in the Usenet newsgroup `comp.compilers` contain much folk wisdom and personal experience reports.

Probably the most complete current treatment of interpreters is [30]. It also contains an extensive bibliography. Another book that contains several articles on interpreter efficiency is [31]. Most of the published literature on interpreters concentrates on decoding speed [4, 32], semantic content, VM design and time/space tradeoffs [32, 33]. There are also some more recent papers about interpreters, discussing issues such as combining VM instructions [34, 16, 3], software pipelining interpreters [3], stack caching [15], improving BTB prediction accuracy [14] and various optimizations [9, 17].

8. Conclusion

Efficient interpreters execute a large number of indirect branches (up to 13% in our benchmarks); without indirect branch prediction, the resulting mispredictions can take most of the time even on a processor with a short pipeline (up to 61%, and more with a longer pipeline).

The commercially available branch prediction mechanisms perform badly (0%–19% accuracy) if the interpreter uses a single, shared dispatch routine for all VM instructions (e.g., with switch dispatch on many compilers). With one copy of the dispatch routine for each VM instruction (e.g., with threaded code), these branch predictors perform somewhat better (37%–50% accuracy for BTBs, 12%–29% accuracy for profile-guided prediction). Two-level indirect branch predictors perform quite well (usually $>$ 90% accuracy), but are not yet implemented in widely available CPUs.

Interpreters using threaded code are up to twice as fast as interpreters using switch dispatch, because switch dispatch executes more native instructions and because its prediction accuracy on BTBs is worse. Further ways to increase the performance of interpreters are: Replicating VM instructions produces better BTB prediction accuracy; combining sequences of VM instructions into superinstructions reduces the number of BTB mispredictions and the number of indirect branches. These techniques can be combined, providing good speedups on processors like the Pentium-III, Athlon, and Pentium 4 and their relatives.

Architectural mechanisms like profile-guided branch prediction and prepare-to-branch instructions can help in theory, but tend to suffer from lack of compiler support in practice (in particular GCC). GCC is the compiler of choice when writing efficient interpreters, because its labels-as-values extension makes it possible to use threaded code.

Acknowledgements

Vitor Santos Costa was a great help in getting Yap to run on SimpleScalar. The referees of Asplos '00, CC '01, Euro-Par '01, and JILP provided many helpful comments on earlier versions of this paper.

References

[1] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy, "The structure and performance of interpreters," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pp. 150–159, 1996.

[2] M. A. Ertl and D. Gregg, "The behaviour of efficient virtual machine interpreters on modern architectures," in *Euro-Par 2001*, pp. 403–412, Springer LNCS 2150, 2001.

[3] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Wiel, "A code compression system based on pipelined interpreters," *Software—Practice and Experience*, vol. 29, pp. 1005–1023, Sept. 1999.

[4] J. R. Bell, "Threaded code," *Communications ACM*, vol. 16, no. 6, pp. 370–372, 1973.

[5] M. A. Ertl, "A portable Forth engine," in *EuroFORTH '93 conference proceedings*, (Mariánské Láznè (Marienbad)), 1993.

[6] R. B. Dewar, "Indirect threaded code," *Communications ACM*, vol. 18, pp. 330–331, June 1975.

[7] X. Leroy, "The ZINC experiment: an economical implementation of the ML language," Technical report 117, INRIA, 1990.

[8] R. A. Kelsey and J. A. Rees, "A tractable Scheme implementation," *Lisp and Symbolic Computation*, vol. 7, no. 4, pp. 315–335, 1994.

[9] V. Santos Costa, "Optimising bytecode emulation for Prolog," in *LNCS 1702, Proceedings of PPDP'99*, pp. 261–267, Springer-Verlag, September 1999.

[10] K. Driesen and U. Hölzle, "Accurate indirect branch prediction," in *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, pp. 167–178, 1998.

[11] D. C. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.

[12] B. Appelbe, R. Das, and R. Harmon, "Future branches – beyond speculative execution," in *Computer Architecture 98 (ACAC '98)* (J. Morris, ed.), vol. 20 of *Australian Computer Science Communications*, (Perth), pp. 1–13, Springer, 1998.

[13] M. Evers, P.-Y. Chang, and Y. N. Patt, "Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches," in 23^{rd} *Annual International Symposium on Computer Architecture*, pp. 3–11, 1996.

[14] M. A. Ertl and D. Gregg, "Optimizing indirect branch prediction accuracy in virtual machine interpreters," in *SIGPLAN '03 Conference on Programming Language Design and Implementation*, 2003.

[15] M. A. Ertl, “Stack caching for interpreters,” in *SIGPLAN ’95 Conference on Programming Language Design and Implementation*, pp. 315–327, 1995.

[16] I. Piumarta and F. Riccardi, “Optimizing direct threaded code by selective inlining,” in *SIGPLAN ’98 Conference on Programming Language Design and Implementation*, pp. 291–300, 1998.

[17] M. A. Ertl, D. Gregg, A. Krall, and B. Paysan, “vmgen — a generator of efficient virtual machine interpreters,” *Software—Practice and Experience*, vol. 32, no. 3, pp. 265–294, 2002.

[18] K. Driesen and U. Hölzle, “The cascaded predictor: Economical and adaptive branch target prediction,” in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-31)*, pp. 249–258, 1998.

[19] K. Driesen and U. Hölzle, “Multi-stage cascaded prediction,” in *EuroPar’99 Conference Proceedings*, vol. 1685 of *LNCS*, pp. 1312–1321, Springer, 1999.

[20] J. Kalamatianos and D. R. Kaeli, “Predicting indirect branches via data compression,” in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-31)*, pp. 272–284, 1998.

[21] J. Kalamatianos and D. Kaeli, “Indirect branch prediction using data compression techniques,” *Journal of Instruction Level Parallelism*, Dec. 1999.

[22] J. Kalamatianos and D. R. Kaeli, “Improving the accuracy of indirect branch prediction via branch classification,” in *Workshop on the Interaction between Compilers and Computer Architectures (INTERACT-3) held in conjunction with ASPLOS-VIII*, 1998.

[23] P.-Y. Chang, E. Hao, and Y. N. Patt, “Target prediction for indirect jumps,” in 24*th* *Annual International Symposium on Computer Architecture*, pp. 274–283, 1997.

[24] A. Krall, “Improving semi-static branch prediction by code replication,” in *Conference on Programming Language Design and Implementation*, vol. 29(7) of *SIGPLAN*, (Orlando), pp. 97–106, ACM, 1994.

[25] C. Young and M. D. Smith, “Improving the accuracy of static branch prediction using branch correlation,” in *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pp. 232–241, 1994.

[26] T.-Y. Yeh and Y. N. Patt, “Two-level adaptive training branch prediction,” in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, (Albuquerque, New Mexico), pp. 51–61, ACM SIGMICRO and IEEE Computer Society TC-MICRO, Nov. 18–20, 1991.

[27] S.-T. Pan, K. So, and J. T. Rahmeh, “Improving the accuracy of dynamic branch prediction using branch correlation,” in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS-V)*, pp. 76–84, 1992.

[28] I.-C. K. Chen, J. T. Coffey, and T. N. Mudge, “Analysis of branch prediction via data compression,” in *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pp. 128–137, 1996.

[29] A. K. Uht, V. Sindagi, and S. Somanathan, “Branch effect reduction techniques,” *IEEE Computer*, vol. 26, pp. 71–81, May 1997.

[30] E. H. Debaere and J. M. Van Campenhout, *Interpretation and Instruction Path Coprocessing*. The MIT Press, 1990.

[31] G. Krasner, ed., *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, 1983.

[32] P. Klint, “Interpretation techniques,” *Software—Practice and Experience*, vol. 11, pp. 963–973, 1981.

[33] T. Pittman, “Two-level hybrid interpreter/native code execution for combined space-time efficiency,” in *Symposium on Interpreters and Interpretive Techniques (SIGPLAN ’87)*, pp. 150–152, 1987.

[34] T. A. Proebsting, “Optimizing an ANSI C interpreter with superoperators,” in *Principles of Programming Languages (POPL ’95)*, pp. 322–332, 1995.