Optimizing Interpreters for Processors with Branch Target Buffers

M. Anton Ertl*, TU Wien
David Gregg, Trinity College Dublin

2002-03-16

Abstract

Interpreters designed for efficiency execute a huge number of indirect branches and can spend more than half of the execution time in indirect branch mispredictions. Branch target buffers are the best widely available form of indirect branch prediction; they produce 50%–100% mispredictions for existing interpreters. In this paper we investigate three methods for improving the prediction accuracy of interpreters: replicating virtual machine (VM) instructions, combining sequences of VM instructions into superinstructions, and using separate dispatch branches for the two outcomes of conditional VM machine branches. In their extreme form these techniques eliminate all mispredictions except those caused by VM-level indirect branches. Applying these techniques in a more conservative way reduces the mispredictions to about 20%. We have measured speedups by factors of 1.75–3.16 on current processors from these techniques.

1 Introduction

Different programming language implementation approaches provide different tradeoffs with respect to the following criteria:

*Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; anton@mips.complang.tuwien.ac.at

- Ease of implementation
- Portability (Retargetability)
- Compilation Speed
- Execution Speed

Interpreters are a popular language implementation approach that can be very good at the first three criteria, but has an execution speed disadvantage: an interpreter designed for efficiency typically suffers a factor of ten slowdown for general-purpose programs over native code produced by an optimizing compiler [HATvdW99].[1] In this paper we investigate how to improve the execution speed of interpreters.

Existing efficient interpreters perform a large number of indirect branches (up to 13% of the executed instructions). Mispredicted branches are expensive on modern processors (e.g., they cost about 10 cycles on the Pentium III and Athlon and 20 cycles on the Pentium 4). As a result, interpreters can spend more than half of their execution time recovering from indirect branch mispredictions [EG01]. Consequently, improving the indirect branch prediction accuracy has a large effect on interpreter performance.

The best indirect branch predictor in widely available processors is the branch target buffer (BTB). Many current desktop and server processors have a BTB or similar structure: Pentium–Pentium 4, Athlon, Alpha 21264. BTBs mispredict 50%–63% of the executed indirect branches in threaded-code interpreters and 81%–98% in switch-based interpreters [EG01].

In this paper, we look at software ways to improve the prediction accuracy. The original contributions in this paper are: 1) We introduce virtual machine instruction replication for increasing the prediction accuracy (Section 4.2). 2) We empirically evaluate how replication and other techniques affect BTB and instruction cache performance (Section 6). 3) We also present a method to compute the expected number of conflict misses in a direct-mapped BTB or cache (Section 5.2).

[1] For library-intensive special-purpose programs the speed difference is usually much smaller. Not all interpreters are designed for efficiency on general-purpose programs and some may produce slowdowns by a factor > 1000 [RLV+96]. Unfortunately, many people draw incorrect general conclusions about the performance of interpreters from such examples.

2 Background

2.1 Efficient Interpreters

This section discusses how efficient interpreters are implemented. We do not have a precise definition for *efficient interpreter*, but 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 are divided into a front-end that compiles the program into an intermediate representation, and an interpreter for that intermediate representation; this design also helps modularity. This paper deals with the efficiency of the interpreter; the efficiency of the front-end can be improved with the established methods for speeding up compiler front-ends.

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.

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 6.3) can consume most of the run-time of an interpreter, so this paper focuses on dispatch.

Dispatching the next VM instruction requires executing one indirect branch to get to the native code that implements the next VM instruction. In efficient interpreters the machine code for simple VM instructions can take as few as 3 native instructions (including the indirect jump), resulting in

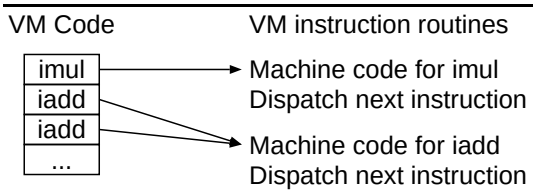
[2]The term *virtual machine* is used in a number of slightly different ways by various people; we use the meaning in the first item of http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?virtual+machine.

```
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 1: VM instruction dispatch using `switch`

a high proportion of indirect branches in the executed instruction mix (we have measured up to 13% for the Gforth interpreter and 11% for the Ocaml interpreter).

There are two popular VM instruction dispatch techniques:




```
GNU C              Alpha assembly
next_inst = *ip;   ldq  s2,0(s1)   ;load next VM instruction
ip++;              addq s1,0x8,s1  ;increment VM instruction pointer
goto *next_inst;   jmp  (s2)       ;jump to next VM instruction
```

Figure 2: Threaded code: VM code representation and instruction dispatch

Switch dispatch uses a large `switch` statement, with one case for each instruction in the virtual machine insturction set. Switch dispatch can be implemented in ANSI C (see Fig. 1), but is not very efficient.

Threaded code represents a VM instruction as address of the routine that implements the instruction [Bel73]. In threaded code the code for dispatching the next instruction consists of fetching the VM instruction, jumping to the fetched address, and incrementing the instruction pointer. This technique cannot be implemented in ANSI C, but it can be implemented in GNU C using the labels-as-values extension. Figure 2 shows threaded code and the instruction dispatch sequence. Threaded code dispatch executes fewer instructions, and provides better branch prediction (see Section 3).

2.2 Branch Target Buffers

CPU pipelines have become longer over time, in order to support faster clock rates and out-of-order superscalar execution. Such CPUs execute straight-line code very fast; however, they have a problem with branches, because they are typically resolved very late in the pipeline (stage n), but they affect the start of the pipeline. Therefore, the following instructions have to proceed through the pipeline for n cycles before they are at the same stage they would be if there was no branch. We can say that the branch takes n cycles to execute (in a simplified execution model).

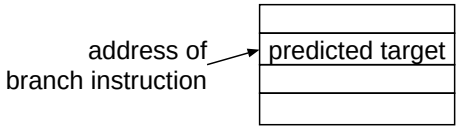


Figure 3: Branch Target Buffer (BTB)

To reduce the frequency of this problem, modern CPUs use branch prediction and speculative execution; if they predict the branch correctly, the branch takes little or no time to execute. The n cycles delay for incorrectly predicted branches is called the misprediction penalty. The misprediction penalty is about 10 cycles on the Pentium III, Athlon, and 21264, and about 20 cycles on the Pentium 4.

The best predictor for indirect branches in widely available CPUs is the branch target buffer (BTB). An idealised BTB contains one entry for each branch and predicts that the branch jumps to the same target as the last time it was executed (see Fig. 3). The size of real BTBs is limited, resulting in capacity and conflict misses. Many current CPUs have a BTB-style predictor, e.g. Pentium–Pentium 4, Athlon, Alpha 21264.

Better indirect branch predictors have been proposed [DH98a, DH98b, DH99, KK98, KK99], but have not been implemented yet in widely available hardware, and it is not clear, if and when they will be available. We expect that optimizations targeting BTBs will be useful for at least the next decade (even BTBs are not universally available yet).

3 Interpreters and BTBs

Ertl and Gregg investigated the performance of several virtual machine interpreters on several branch predictors [EG01] and found that BTBs mispredict 81%–98% of the indirect branches in switch-dispatch interpreters, and 57%–63% of the indirect branches in threaded-code interpreters (a variation, the so-called BTB with two-bit counters, produces slightly better results for threaded code: 50%–61% mispredictions).

What is the reason for the differences in prediction accuracy between the two dispatch methods? The decisive difference between the dispatch methods is this: A copy of the threaded code dispatch sequence is usually appended to the native code for each VM instruction; as a result, each VM instruction

VM program	switch dispatch BTB entry	next instruction prediction	actual	threaded code BTB entry	next instruction prediction	actual
label:						
A	switch	A	B	br-A	LOOP	B
B	switch	B	A	br-B	A	A
A	switch	A	GOTO	br-A	B	GOTO
GOTO label	switch	GOTO	A	br-GOTO	A	A

Figure 4: BTB predictions on a small VM program

has its own indirect branch. In contrast, with switch dispatch all compilers we have tested produce a single indirect branch (among other code) for the `switch`, and they compile the `break`s into unconditional branches to this common dispatch code. In effect, the single indirect branch is shared by all VM instructions.

Why do these mispredictions occur? Consider the VM code fragment in Fig. 4, and imagine that the loop has been executed at least once.

With switch dispatch, there is only one indirect branch, the switch branch, and consequently there is only one BTB entry involved. When jumping to the native code for VM instruction A, the BTB entry is updated to point to that native code routine. When the next VM instruction is dispatched, the BTB will therefore predict target A; in our example the next instruction is B, so the BTB mispredicts. The BTB now updates the entry for the switch instructions to point to B, etc. So, with switch dispatch the BTB always predicts that the current instruction will be executed next, which is rarely correct.

For threaded code, each VM instruction has its own indirect branch and BTB entry (assuming there are no conflict or capacity misses in the BTB); e.g., instruction A has Branch br-A and BTB entry br-A, etc. So, when VM instruction B dispatches the next instruction, the same target will be selected as on the last execution of B; since B occurs only once in the loop, the BTB will always predict the same target: A. Similarly, the branch of the GOTO instruction will also be predicted correctly (branch to A). However, A occurs twice in our code fragment, and the BTB always uses the last target for the prediction (alternatingly B and GOTO), so the BTB will never predict A's dispatch branch correctly.

```
IF_NULL:
branch_offset = *ip++;
if ( top_of_stack == NULL ) {
  ip = ip + branch_offset;
  goto *ip++;
}
else {
  goto *ip++;
}
```

Figure 5: Replicating the dispatch code in VM branch instructions

We will concentrate on interpreters using separate dispatch branches in the rest of the paper.

4 Improving the Prediction Accuracy

Generally, as long as a VM instruction occurs only once in the working set of the interpreted program, the BTB will predict its dispatch branch correctly, because the instruction following the VM instruction is the same on all executions. But if a VM instruction occurs several times, mispredictions are likely.

The exception to this rule are conditional and indirect VM branch instructions; for these VM instructions the next executed VM instruction can vary, leading to possible mispredictions even when the VM instruction occurs only once in the working set.

4.1 Conditional VM Branches

Conditional VM branches have two possible following VM instructions, one on the fall-through path, and one on the branch-taken path. By using different indirect branches for these two cases we can make the dispatch of conditional VM branches as predictable as the dispatch of regular, non-branching VM instructions. This optimization can be easily performed by moving the dispatch code up into the if-statement that updates the VM instruction pointer (see Fig. 5).

```
        IF_NULL label
        A
label:  B
```

Figure 6: Branching example

VM program	BTB entry	threaded code next instruction prediction	actual
label:			
A_1	br-A_1	B	B
B	br-B	A_2	A_2
A_2	br-A_2	GOTO	GOTO
GOTO label	br-GOTO	A_1	A_1

Figure 7: Improving BTB prediction accuracy by replicating VM instructions

Figure 6 shows a small section of VM code containing a branch. Let us assume that this VM code section is executed many times, and that the branch condition is true every second time the code is executed. Thus, on half the executions, the next VM instruction to be executed after IF_NULL is A, and on the other half B. If the implementation of IF_NULL contains only a single dispatch routine, then the BTB entry for the indirect branch in that routine will always contain the wrong value. Where it should predict B, it will predict A, and vice versa. On the other hand, if there are separate dispatch routines, as shown in Fig. 5, there will be separate indirect branches, and thus separate BTB entries. In this case, the BTB will always predict the dispatch branches correctly (except for BTB misses).

4.2 Replicating VM Instructions

In order to avoid having the same VM instruction several times in the working set, we can create several versions of the same instruction. We copy the code for the VM instruction, and use different copies in different places. If a copy occurs only once in the working set, its branch will predict the next instruction correctly.

Figure 7 shows how replication works in our example. There are two

copies of the VM instruction A now, A_1, and A_2. Each of these copies has its own dispatch branch and its own entry in the BTB. Because A_1 is always followed by B, and A_2 is followed by GOTO, the dispatch branches of A_1 and A_2 always predict correctly, and there are no mispredictions after the first iteration while the interpreter executes the loop (except possibly mispredictions from capacity or conflict misses).

There are two ways to implement this replication optimization:

Static: The static approach produces copies of popular VM instructions at interpreter build time. The interpreter's front end chooses one of the available copies of a VM instruction each time it generates that VM instruction.

Two plausible ways to choose the copy come to mind: round-robin (i.e., always choose the statically least-recently-used copy) and random. Round-robin should work better if the static code generation order correlates with the dynamic execution order (it does so at least within basic blocks); we tried both approaches in our simulator, and achieved better results for round-robin, so we use that in the rest of the paper.

Dynamic: The front end of the interpretive system generates the copies during VM code generation. I.e., it copies the machine code that implements a VM instruction and lets the threaded code pointer point to the new copy. In this way each static occurence of the VM instruction gets its own copy.

To avoid getting too much code growth and the resulting instruction cache misses, the front end might reuse some copies for several static occurences of VM instructions. However, if several reuses of the same copy occur in the working set, this may help the I-cache, but it will probably result in additional mispredictions; conversely, if no such reuses occur in the working set, the reuses may reduce the overall code size, but will not reduce I-cache misses.

One problem with the dynamic approach is that it can only copy code that is relocatable; i.e., it cannot copy code, if the code fragment contains a PC-relative reference to something outside the code fragment (e.g., a 386 call instruction), or if it contains an absolute reference to something inside the code fragment (e.g., a MIPS j(ump) instruction).

	threaded code		
VM	BTB	next instruction	
program	entry	prediction	actual
label:			
A	br-A	B_A	B_A
B_A	br-B_A	GOTO	GOTO
GOTO label	br-GOTO	A	A

Figure 8: Improving BTB prediction accuracy with superinstructions

The disadvantages of the static approach are: It requires substantial memory and time during the compilation of the interpreter to provide many copies, limiting the practical number of additional copies to a few thousand. These copies are selected once for all programs, whereas a different selection may be better for a specific interpreted program.

The disadvantages of the dynamic approach are: It requires (a small amount of) platform-specific code, in particular cache-flushing code, but possibly other special features (e.g., on MIPS it might have to ensure that the copies are in the same 256MB region as the original code to ensure that the J and JAL instructions continue to work). And it cannot copy non-relocatable VM instructions.

4.3 Superinstructions

Combining several VM instructions into superinstructions is a technique that has been used for reducing VM code size and for reducing the dispatch and argument access overhead in the past [Pro95, PR98, HATvdW99].

In this paper we investigate the effect of superinstructions on dispatch mispredictions; in particular, we find that using superinstructions reduces mispredictions far more than it reduces dispatches or executed native instructions (see Section 6.3).

In Figure 8 we have combined the sequence B A into the superinstruction B_A. This superinstruction occurs only once in the loop, and A now also occurs only once, so there are no mispredictions after the first iteration while the interpreter executes the loop.

There are two approaches to using superinstructions:

Static: The static approach (used in, e.g., vmgen [EGKP02]) determines

the available superinstructions at interpreter build time; the interpreter writer selects a number of superinstructions that are useful for many programs and implements them in the VM interpreter (possibly with automatic support). The interpreter's front end combines the appropriate sequences of VM instructions into these superinstructions.

Dynamic: The dynamic approach [PR98] generates the superinstructions needed for the particular interpreted program in the front end of the interpretive system. The front end copies the machine code for each component instruction of the superinstruction, concatenating them to form the machine code for the superinstruction; the dispatch code of the component instructions is left away, except for the last instruction in the superinstruction. Piumarta and Riccardi proposed combining whole VM-level basic blocks into superinstructions.

The disadvantages of static and dynamic replication also apply to static and dynamic superinstructions. In addition, these approaches have the following advantages:

The static approach enables machine code optimizations crossing component instruction boundaries; in particular, much of the overhead of accessing VM instruction arguments can be eliminated, and the native code for the superinstruction can be scheduled for best performance.

An advantage of the dynamic approach is that the superinstructions fit the interpreted program exactly, so fewer superinstructions are executed.

4.4 Combining the optimizations

Combining these three optimizations reveals a few synergies:

- It is actually easier to use dynamic superinstructions with dynamic replication than to use dynamic superinstructions without replication. Eliminating additional copies of the superinstrution takes an additional step; this step was proposed to conserve program memory and reduce I-cache misses [PR98], but our results indicate that it is not a good idea on processors with BTBs.

- Using two dispatches for conditional VM branches makes it easy to extend superinstructions beyond conditional branches: The fall-through dispatch must be arranged as the last part of the VM instruction; then

the dispatch can be replaced by the code for another instruction just as usual; the branch-taken dispatch is not eliminated and is executed when necessary.

5 Experimental Setup

5.1 The Simulator

In addition to building, running and timing interpreters with some of these optimizations, we used a simulation approach; the advantages are:

- We can simulate a variety of BTB sizes, including the ideal (infinite) BTB.
- We avoid interference from such mostly-random effects as scheduling variations, optimization variations (e.g., register spills), cache and BTB conflicts.
- We can simulate interpreters that we cannot build on our machines due to memory constraints (e.g., more than 2000 static superinstructions).
- The simulator is easier to implement and use, allowing us to evaluate more variations.

The disadvantage of our simulation approach is that the results are only indirect metrics such as mispredictions and instruction cache misses instead of the metric of interest: execution time. Therefore, we also implemented these techniques in an interpreter and used it to produce timing results (see Section 6.3).

We built the simulator by adding instrumentation code to the Gforth interpreter. Every time the interpreter executes a VM instruction, it calls a routine that simulates the effect of this VM instruction on the BTB and the instruction cache. The interpreter passes the VM instruction pointer and a branch taken/not-taken flag (for conditional branches) to the simulation routine.

To simulate VM instruction replication, the simulation actually keeps a shadow copy of the executed program where the opcodes of the VM instructions are replaced by IDs of the various copies of the VM instruction. Similarly, superinstructions are simulated by having an ID for the superinstruc-

tion as shadow copy of the last VM instruction, and IDs for "no instruction" as shadow copies of the other instructions in the superinstruction.

We validate the simulator by comparing the number of simulated mispredictions of the *Brainless* benchmark to the number of *taken mispredicted* events on the Athlon when running the benchmark with the original Gforth interpreter. The results are very close to each other: the Athlon counted 1.1% more mispredicted taken branches than our simulator counted dispatch branch mispredictions; the explanation for this difference is that there are taken branches that are not dispatch branches (mainly conditional branches).

5.2 Expected Misses

All of our optimizations increase the number of indirect branches and the code size. Therefore, the number of capacity and conflict misses in finite BTBs and instruction caches is also likely to increase, and this increase might compensate some of the increased prediction accuracy.

In order to quantify these effects, we also want to produce the number of expected misses. These structures typically have a low associativity, and simulating such a structure directly would introduce mostly-random effects from conflict misses and such effects could mask some of the more subtle effects in the results.

Therefore, we compute a statistical number of expected misses in a direct-mapped BTB or cache. The probability of a hit in a direct-mapped BTB or cache with n entries/lines is $((n-1)/n)^k$ if k different entries have been accessed since the last time the item of interest was accessed.

So, the simulator maintains a move-to-front array of the most recent accesses, and another array R that counts how often the k^{th} most recent item was accessed. In the end the simulator can compute the expected number of hits for a structure of size n like this:

$$\sum_k R_k \left(\frac{n-1}{n}\right)^k$$

5.3 Benchmarks

We use the following programs as benchmarks: *Gray*, a parser generator processing an Oberon grammar; *Vmgen*, an interpreter generator processing the Gforth VM; and *Bench-gc*, a program exercising a conservative garbage

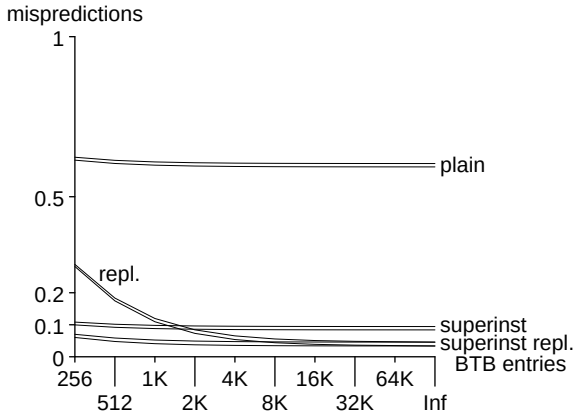


Figure 9: Mispredictions per original dispatch for *Bench-gc*

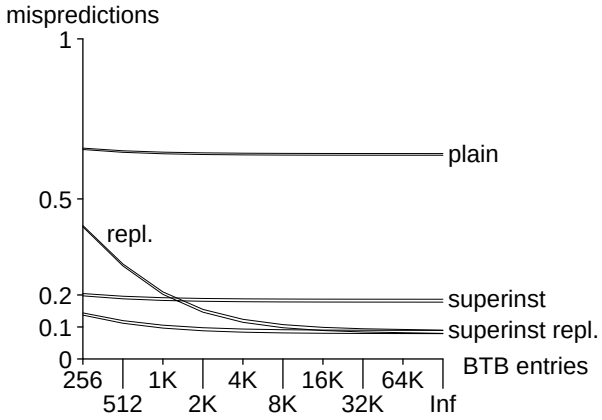


Figure 10: Mispredictions per original dispatch for *Gray*

collector. We used *Brainless*, a chess program, as training program for static replication and static superinstructions.

6 Results

6.1 Limits

Figures 9, 10, and 11 show how the eight possible combinations of our three optimizations (optimizing conditional VM branches, replication, and using

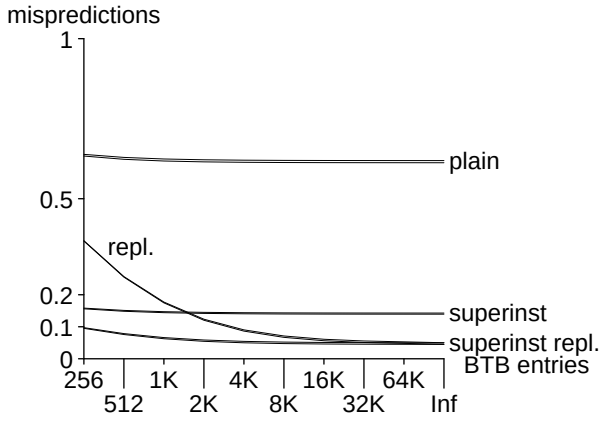


Figure 11: Mispredictions per original dispatch for *Vmgen*

superinstructions) affect the branch prediction accuracy. We chose the extreme forms of replication and superinstructions for this experiment:

Each occurence of a VM instruction in the interpreted program uses a separate copy of that VM instruction. Combined with the conditional VM branch optimization this should eliminate all mispredictions except those caused by indirect VM branches. And indeed it does; the remaining mispredictions (3.4%–7.9% in our benchmarks) are nearly all caused by VM procedure returns (i.e., an indirect VM branch)[3].

For superinstructions we combined each basic block into a superinstruction, just like the dynamic method would.

In Figs. 9–11 you see four pairs of lines; the difference between the two lines of each pair is the use or non-use of the conditional VM branch optimization. The conditional VM branch optimization results in a small improvement in mispredictions independent of the other optimizations. We look only at configurations with this optimization in the rest of this paper.

Code replication produces the least number of mispredictions for large BTBs, but produces a large number of conflict misses for small BTBs, unless superinstructions are used, too. Current BTB sizes are 512 entries (Pentium–Pentium III), 1024 entries (Alpha 21264), 2048 entries (Athlon), and 4096 entries (Pentium 4).

[3]This large proportion of mispredictions in returns is caused by the large number of executed returns in our benchmarks; in Forth code, 12%–17%of the dynamically executed instructions are returns [Ert95].

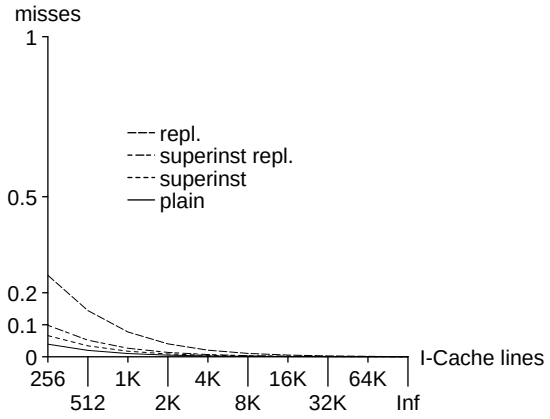


Figure 12: I-cache misses per original dispatch for *Bench-gc*

Superinstructions improve the prediction accuracy over the original interpreter significantly, but replication is better for large BTBs.

The minimum number of mispredictions is achieved by combining all three optimizations.

However, we also have to consider the change in I-cache misses. Figure 12 gives a rough idea of the I-cache misses for *Bench-gc*; this figure assumes that each VM instruction or superinstruction occupies exactly one cache line.

Typical I-caches today have 512 (Pentium III) or 1024 lines (Athlon, 21264). An I-cache miss that hits the L2 cache costs about the same as a branch misprediction (around 10 cycles). An I-cache line (32 bytes on Pentium III, 64 bytes on Athlon and 21264) can typically contain 2-3 simple VM instructions and correspondingly fewer superinstructions, so in reality we will probably see fewer misses for Plain and Repl than shown in Fig. 12.

We see that I-cache misses can be a significant problem with small I-caches, in particular for the two replication variants. However, if we also consider the improvement in prediction accuracy, replication-based approaches still work better than their non-replicating counterparts for the benchmarks and configurations we used.

6.2 Static methods

To reach the limits explored above the interpreter must be able to copy all virtual machine instructions at run-time and/or combine them into superin-

structions. This may not be possible for some VM instructions, and it may not be desirable for portability reasons. What can we achieve with methods that work at interpreter generation time?

We measured three basic configurations (all with the conditional VM branch optimization):

Repl. Have n copies of VM instructions that are executed k times in the training run, $n = 1 + \lfloor k/c \rfloor$. Various constants c result in interpreters with varying numbers of additional VM instructions. In the front-end we use a round-robin scheme for selecting the copy to use when compiling a VM instruction.

Superinst The m most frequent sequences of VM instructions in the training run become superinstructions.

Superinst repl. A combination of Superinst and Repl.: Do a training run with a Superinst interpreter and then replicate normal VM instructions and superinstructions like in Repl. We use a balance of about 5:3 between (original) superinstructions and replicas in our experiments.

Figure 13 shows how these methods perform on *Gray* with an unlimited BTB (the other benchmarks perform similarly). Superinst performs relatively well with few additional VM instructions, but Repl. dominates if many additional VM instructions are available. Superinst repl. takes a midway position.

Our explanation for the domination of Repl. over the Superinst-based methods in the right part of Fig. 13 is that the benefit of Repl. transfers better between the training program and the reference program, whereas the superinstructions we selected were apparently too specific to the training program. Using several training programs should avoid that.

Overall, with several thousand additional VM instructions the mispredictions can be reduced to 6%–13% of the original dispatches. With a more realistic setup of using several hundred additional VM instructions, there are 13%–26% mispredictions.

The dominance of Repl. reverses itself on small BTBs (see Fig. 14), where it causes too many conflict misses. Here Superinst repl. dominates.

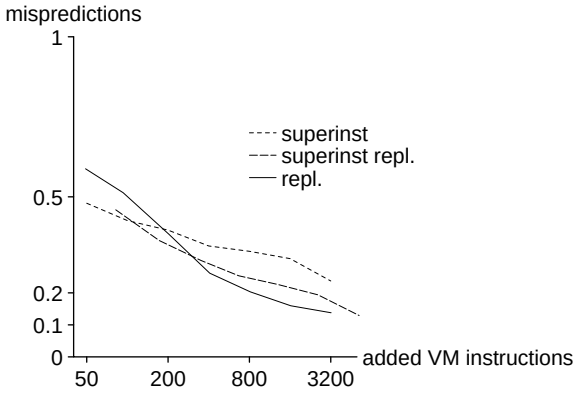


Figure 13: Static methods: mispredictions of an unlimited BTB per original dispatch for *Gray*

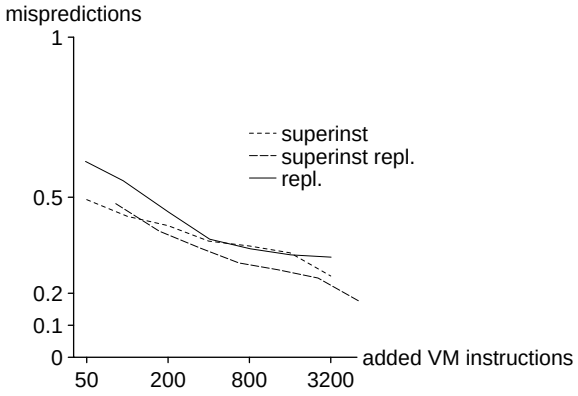


Figure 14: Static methods: mispredictions of a 512-entry BTB per original dispatch for *Gray*

6.3 Real world results

We have implemented some of these techniques in Gforth, and compare the following variants:

Original: Classical threaded code, with separate dispatch routines for the two outcomes of conditional branches.

Static superinsts: Original, with 400 static superinstructions from a *brainless* training run added.

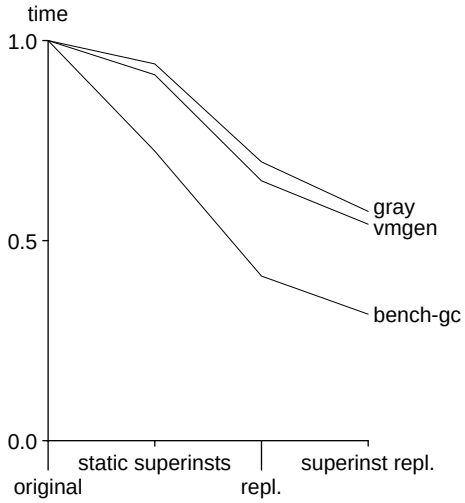


Figure 15: Relative execution time of various techniques on a Celeron

Repl.: Original with dynamic replication.

Superinst repl.: Original with dynamic superinstructions and replication.

Figure 15 shows the run-time of various benchmarks on a Celeron; the results on an Athlon are similar. Both dynamic techniques provide more speedup than static superinstructions. For *Gray* and *Vmgen* the small speedup of static superinstructions is mostly caused by a current limitation in the implementation: currently the static superinstruction optimization is not applied to library code, and these benchmarks spend a lot of time in library code.

Figure 16 shows the reason for these speedups: they are caused mainly by better branch prediction, and to a smaller part by a reduction in executed instructions. In particular, *Repl* executes the same number of instructions and branches (just different copies), so the factor 2.41 of speedup is exclusively from better branch prediction. Once the mispredictions are under control, reducing the number of executed instructions can provide further speedups.

We also took a look at the number of I-cache misses, but they were negligible (at worst (repl.) 68000 I-cache misses compared to more than 2000000 remaining mispredictions). This is better than expected from the simulation. The reasons for this difference are: several VM instructions fit in each cache line, putting less pressure on the cache than assumed by the

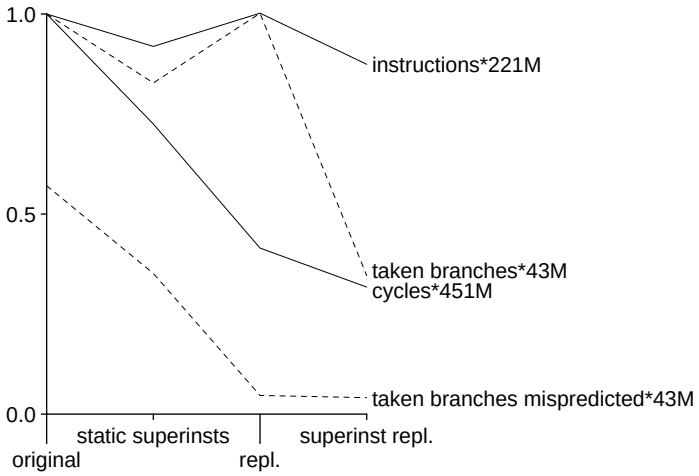


Figure 16: Performance-relevant events during *Bench-gc* execution

simulation; the I-cache is four-way set associative, whereas the simulation assumed a direct-mapped cache; and there is spatial locality between the instructions that the simulation does not take into account.

7 Related work

The accuracy of static conditional branch predictors has been improved with software methods: branch alignment [CG94] and code replication [Kra94, YS94, YGS95]. The present paper looks at using software methods to improve the accuracy of the BTB, a simple dynamic indirect branch predictor.

Better indirect branch predictors than BTBs have been proposed in a number of papers [DH98a, DH98b, DH99, KK98, KK99] and they work well on interpreters [EG01], but they are not available in hardware yet, and it will probably take a long time before they are universally available.

There are a number of recent papers on improving interpreter performance [Pro95, Ert95, PR98, SC99]. Software pipelining the interpreter [HA00, HATvdW99] is a way to reduce the branch dispatch costs on architectures with delayed indirect branches (or split indirect branches).

Ertl and Gregg [EG01] investigated the performance of various branch predictors on interpreters, but did not investigate means to improve the prediction accuracy beyond threaded code.

Papers dealing with superoperators and superinstructions [Pro95, PR98, HATvdW99, EGKP02] concentrated on reducing the number of executed dispatches and sometimes the VM code size, but have not evaluated the effect of superinstructions on BTB prediction accuracy (apart from two paragraphs in [EGKP02]). In particular, Piumarta and Riccardi invested extra work to avoid replication (in order to reduce code size), but this increases mispredictions on processors with BTBs.

8 Conclusion

If a VM instruction occurs several times in the working set of an interpreted program, a BTB will frequently mispredict the dispatch branch of the VM instruction. We present three techniques for reducing mispredictions in interpreters: using two indirect branches for the two outcomes of a conditional VM branch (easy to implement, but gives only minor benefits); replicating VM instructions, such that hopefully each replica occurs only once in the working set; and combining sequences of VM instructions into superinstructions.

In their extreme form the combination of these techniques work very well and eliminate all mispredictions except those from indirect branches. Replication on its own may cause many conflict misses in small BTBs and small I-caches, but combined with superinstructions it performs well even in such environments. The speedup resulting from the better prediction accuracy alone is 1.44–2.41 on a Celeron, and the total speedup is 1.75–3.16.

This extreme form may not be available for technical or portability reasons, so we also look at alternatives that do not require run-time code generation: With a few hundred replications and superinstructions determined at interpreter build time, the number of mispredictions can be reduced to about 20% of the original dispatches. The speedup from static superinstructions is less impressive than for the dynamic techniques (1.38 for bench-gc and 400 static superinstructions).

The simulator we used will be available at
http://www.complang.tuwien.ac.at/anton/interpreter-btb/.

References

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[CG94] Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 242–251, 1994.

[DH98a] K. Driesen and U. Hölzle. Accurate indirect branch prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, pages 167–178, 1998.

[DH98b] 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)*, pages 249–258, 1998.

[DH99] Karel Driesen and Urs Hölzle. Multi-stage cascaded prediction. In *EuroPar'99 Conference Proceedings*, volume 1685 of *LNCS*, pages 1312–1321. Springer, 1999.

[EG01] M. Anton Ertl and David Gregg. The behaviour of efficient virtual machine interpreters on modern architectures. In *Euro-Par 2001*, pages 403–412. Springer LNCS 2150, 2001.

[EGKP02] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. vmgen — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.

[Ert95] M. Anton Ertl. Stack caching for interpreters. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315–327, 1995.

[HA00] Jan Hoogerbrugge and Lex Augusteijn. Pipelined Java virtual machine interpreters. In *Proceedings of the 9th International Conference on Compiler Construction (CC' 00)*. Springer LNCS, 2000.

[HATvdW99] Jan Hoogerbrugge, Lex Augusteijn, Jeroen Trum, and Rik van de Wiel. A code compression system based on pipelined interpreters. *Software—Practice and Experience*, 29(11):1005–1023, September 1999.

[KK98] 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)*, pages 272–284, 1998.

[KK99] John Kalamatianos and David Kaeli. Indirect branch prediction using data compression techniques. *Journal of Instruction Level Parallelism*, December 1999.

[Kra94] Andreas Krall. Improving semi-static branch prediction by code replication. In *Conference on Programming Language Design and Implementation*, volume 29(7) of *SIGPLAN*, pages 97–106, Orlando, 1994. ACM.

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[Pro95] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Principles of Programming Languages (POPL '95)*, pages 322–332, 1995.

[RLV+96] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The structure and performance of interpreters. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 150–159, 1996.

[SC99] Vítor Santos Costa. Optimising bytecode emulation for Prolog. In *LNCS 1702, Proceedings of PPDP'99*, pages 261–267. Springer-Verlag, September 1999.

[YGS95] Cliff Young, Nicolas Gloy, and Michael D. Smith. A comparative analysis of schemes for correlated branch prediction. In

22^{nd} *Annual International Symposium on Computer Architecture*, pages 276–286, 1995.

[YS94] Cliff Young and Michael D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 232–241, 1994.