Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters

M. Anton Ertl*
TU Wien

David Gregg
Trinity College, Dublin

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; however, their prediction accuracy for existing interpreters is only 2%–50%. In this paper we investigate two methods for improving the prediction accuracy of BTBs for interpreters: replicating virtual machine (VM) instructions and combining sequences of VM instructions into superinstructions. We investigate static (interpreter build-time) and dynamic (interpreter run-time) variants of these techniques and compare them and several combinations of these techniques. These techniques can eliminate nearly all of the dispatch branch mispredictions, and have other benefits, resulting in speedups by a factor of up to 3.17 over efficient threaded-code interpreters, and speedups by a factor of up to 1.3 over techniques relying on superinstructions alone.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.4 [**Programming Languages**]: Processors—*Interpreters*

General Terms

Languages, Performance, Experimentation

Keywords

Interpreter, branch target buffer, branch prediction, code replication, superinstruction

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

PLDI'03, June 9–11, 2003, San Diego, California, USA.

1. INTRODUCTION

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

- 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 [10].[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 [7]. 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). Most current desktop and server processors have a BTB or similar structure: all Pentiums, Athlon, Alpha 21264, Itanium 2. BTBs mispredict 50%–63% of the executed indirect branches in threaded-code interpreters and 81%–98% in switch-based interpreters [7].

In this paper, we look at software ways to improve the prediction accuracy. The main contributions of this paper are:

- We propose the new technique of replication (Section 4.1) for eliminating mispredictions.

[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 [15]. Unfortunately, many people draw incorrect general conclusions about the performance of interpreters from such examples.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in:
SIGPLAN '03 Conference on Programming Language Design and Implementation (PLDI '03)

```
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`

- We evaluate this technique, as well as existing superinstruction techniques, and the combination of these techniques with respect to prediction accuracy and performance (Section 7).
- We introduce several enhancements of dynamic superinstructions (in addition to replication), in particular: extending them across basic blocks; and a portable way to detect non-relocatable code fragments (Section 5.2).
- We empirically compare the static [8] and dynamic [13] superinstruction techniques against each other (Section 7).

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 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 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 [7]).

There are two popular VM instruction dispatch techniques:

Switch dispatch uses a large `switch` statement, with one case for each instruction in the virtual machine instruction set. Switch dispatch can be implemented in ANSI C (see Fig. 1), but is not very efficient [7] (see also Section 3).

Threaded code represents a VM instruction as address of the routine that implements the instruction [1]. 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).

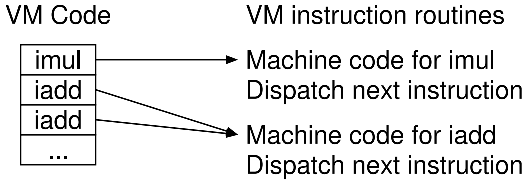
Several interpreters use threaded code when compiling with GCC, and fall back to switch dispatch, if GCC is not available (e.g., the Ocaml interpreter, YAP, Sicstus Prolog).

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).

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

[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.




```
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

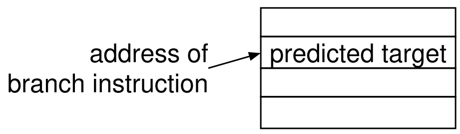


Figure 3: Branch Target Buffer (BTB)

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. Most current CPUs have a BTB-style predictor, e.g. all Pentiums, Athlon, Alpha 21264, Itanium 2.

Better indirect branch predictors have been proposed [4, 5, 11], and they would improve the prediction accuracy in interpreters substantially [7]. However, they have not been implemented yet in widely available hardware, and it is not clear, if and when they will be available. The software techniques explored in this paper improve the prediction accuracy now, by a similar amount.

3. INTERPRETERS AND BTBS

Ertl and Gregg investigated the performance of several virtual machine interpreters on several branch predictors [7] 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 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 also be the next one to be executed, 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.

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.

4.1 Replicating VM Instructions

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

Figure 5 shows how replication works in our example.

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	GOTO	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

VM program	threaded code BTB entry	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 5: Improving BTB prediction accuracy by replicating VM instructions

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

Figure 6: Improving BTB prediction accuracy with superinstructions

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 in the BTB).

4.2 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 [14, 13, 10, 8]. However, its effect on branch prediction has not been investigated in depth yet.

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 7.3).

To get an idea why this is the case, consider Figure 6: we combine 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.

5. IMPLEMENTATION

5.1 Static Approach

There are two ways of implementing replication and superinstructions (see Fig. 7).

In the static approach the interpreter writer produces replicas and/or superinstructions at interpreter build-time, typically by generating C code for them with a macro processor or interpreter generator (e.g., vmgen supports static superinstructions [8]). During VM code generation (at interpreter run-time) the interpreter front-end just selects among the built-in replicas and/or superinstructions.

For static replication two plausible ways to select the copy come to mind: round-robin (i.e., always select the statically least-recently-used copy) and random. We tried both approaches in our simulator, and achieved better results for round-robin, so we use that in the rest of the paper. Our explanation for the better results with round-robin selection is spatial locality in the code; execution does not jump around in the code at random, but tends to stay in a specific region (e.g., in a loop), and there it is less likely to encounter the same replica twice with round-robin selection. E.g., in our example loop we will get the perfect result (Fig. 5) if we have at least two replicas of A and use round-robin selection, whereas random selection might use the same replica of A twice and thus produce 50% mispredictions.

For static superinstructions one can use dynamic programming (shortest-path algorithm) to select the optimal (minimum) number of superinstructions for a given basic block [2]. A simpler alternative is the greedy (maximum munch) algorithm. In the rest of the paper we use the greedy algorithm (because we have not yet implemented dynamic programming); preliminary simulation results indicate there is almost no difference between the results for optimal and greedy selection.

5.2 Dynamic Approach

In the dynamic approach the replicas or superinstructions are created when the VM code is produced at interpreter run-time.

To implement replication, every time the interpreter front-end generates a VM instruction, it creates a new copy of the code for the VM instruction, and lets the threaded code pointer point to the new copy (see Fig. 7). In this way each instance of a VM instruction gets its own replica, ensuring that there are no mispredictions (apart from those resulting from the limited BTB size). The original copies of the code are only used for copying, and are never executed. The front-end knows the end of the code to be copied through a label there [13].

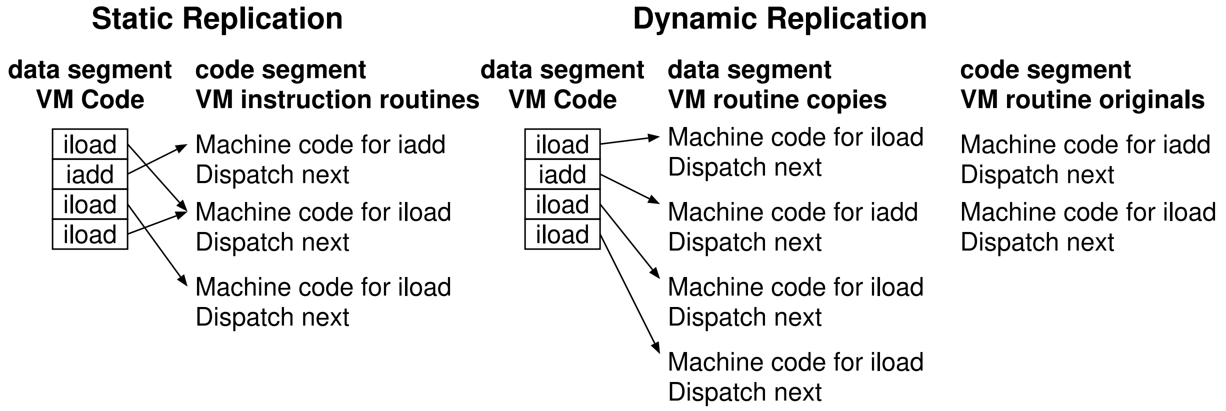


Figure 7: The static and the dynamic approach to implementing replication

Implementing dynamic replication with dynamic superinstructions requires only a small change over dynamic replication alone, if the replicas are already laid down in memory in the same sequence as the VM code: just do not copy the dispatch code except on VM basic block ends. This results in one superinstruction for each basic block.

If you want dynamic superinstructions without replication, you have to perform another change: at the end of each VM basic block, check if the superinstruction has already occured; if so, eliminate the new replica, and redirect the threaded code pointers to the first version of the superinstruction (see [13]).

One can get dynamic superinstructions larger than a basic block with two more changes:

- Keep the increments of the VM instruction pointer even if you do not copy the rest of the dispatch code; as a result, the VM code will be quite similar to the dynamic replication case (whereas you have only one threaded-code pointer per superinstruction if you eliminate the increments); this allows to continue the superinstruction across VM code entry points; on a VM jump to the entry point, the threaded code pointer at this place will be used and result in entering the code for the superinstruction in the middle.
- Let the dispatch for the fall-through path of a conditional VM branch be at the end of the conditional branch code, and use an additional dispatch for the branch-taken path; then you can also eliminate the (fall-through) dispatch at the end of a conditional VM branch.

As a result of these two optimizations, all dispatches are eliminated, except dispatches for taken VM branches, VM calls and VM returns (see Fig. 8).

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., an Intel 386 `call` instruction), or if it contains an absolute reference to something inside the code fragment (e.g., a MIPS `j`(ump) instruction). Whether the code for a VM instruction is relocatable or not depends on the architecture and on the compiler; so, a general no-copying list [13] is not sufficient.

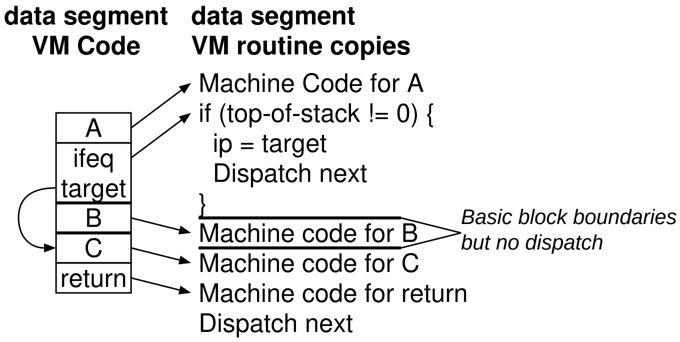


Figure 8: Superinstructions across basic blocks

Our approach to this problem is to have two versions of the VM interpreter function, one with some gratuitious padding between the VM instructions. We compare the code fragments for each VM instruction of these two functions; if they are the same, the code fragment is relocatable, if they are different, it is not.

The dynamic approach requires a small amount of platform-specific code; on most architectures it needs only code for flushing the I-cache, but, 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.

5.3 Comparison

The main advantage of the static approach is that it is completely portable, whereas the dynamic approach requires a small amount of platform-specific code.

Another advantage of static superinstructions is that their code can be optimized across their component instructions, whereas dynamic superinstructions simply concatenate the components without optimization. In particular, static superinstructions can keep stack items in registers across components, and combine the stack pointer updates of the components. In addition, static superinstructions make it possible to use instruction scheduling across component VM instructions. These advantages can also be exploited in a dynamic setting by combining static superinstructions with dynamic superinstructions and dynamic replication.

Moreover, static replication and superinstructions also work for non-relocatable code. However, at least for Gforth the code for the frequently-executed VM instructions is relocatable on the 386 and Alpha architecture. For the JVM, instructions that can throw exceptions are often non-relocatable (relative branch to the throw code outside the code for the VM instruction), but that can be worked around, e.g., by using an indirect branch instead of the relative branch.

Finally, the static approach does not need to pay the cost of copying the code at run-time (including potentially expensive I-cache flushes), that the dynamic approach has to pay. However, in our experiments this copying takes 5ms for a 10000-line program (190KB generated code) on a Celeron-800, so that should usually not be a problem[3].

The main advantage of the dynamic approach is that it perfectly fits replications and/or superinstructions to the interpreted program, whereas the static approach has to select one set of replications/superinstructions for all programs.

Another advantage of the dynamic approach is that the number of replications and superinstructions is only limited by the resulting code size, whereas in the static approach the time and space required for compiling the interpreter limit the number of replications and superinstructions to around 1000 (e.g., compiling Gforth with 1600 superinstructions requires 5 hours and 400 MB on a Celeron-800).

5.4 Relation to just-in-time compilers

The machine code resulting from dynamic superinstructions with replication is similar to what a simple just-in-time (JIT) native-code compiler produces. So why not write a JIT compiler in the first place?

The reason is *portability*. Native-code compilers take a significant effort to retarget to another architecture (typically months to years, for each architecture). In contrast retargeting the dynamic replication/superinstruction part from the 386 to the Alpha architecture took about an hour. And if we do not invest the hour, we can still fall back to the base interpreter on the new architecture; in contrast, if you do not want to invest the months of effort for retargeting a JIT, you need a separate fallback system (e.g., an interpreter), and that needs even more effort. And all these targets and the fallback system have to be maintained, requiring yet more effort.

Technically, the main difference between code from a simple, macro-expanding native-code compiler and our code from dynamic replication with dynamic superinstructions is: our code accesses immediate arguments of VM instruction through the VM code representation; and it uses indirect branches instead of direct branches for control-flow changes.

See Section 7.6 for a timing comparison.

6. EXPERIMENTAL SETUP

We have conducted experiments using a simulator as well as experiments using an implementation of these techniques. We used a simulator to get results for various hardware configurations (especially varying BTB and cache sizes), and to get results without noise effects like cache or BTB conflicts, or (for static methods) instruction scheduling or register allocation differences.

[3]Actually, in comparison to plain threaded code, the copying overhead is already amortized by the speedup of the Forth-level startup code, leading to the same total startup times (17ms on the Celeron-800).

Program	Version	Lines	Description
gray	4	754	parser generator
bench-gc	1.1	1150	garbage collector
tscp	0.4	1625	chess
vmgen	0.5.9	2068	interpreter generator
cross	0.5.9	2735	Forth cross-compiler
brainless	0.0.2	3519	chess
brew	38	29804	evolutionary programming

Figure 9: Benchmark programs used

The results from the simulation and the real implementation agree reasonably well, so in this paper we mainly report results from the implementation running on real processors, and we refer to the simulation results only to clarify points that are not apparent from the real-world implementation results.

6.1 Implementation

We implemented the techniques described in Section 4 in Gforth, a product-quality Forth interpreter.

In particular, we implemented static superinstructions using vmgen [8]; we implemented static replication by replicating the code for the (super)instructions on interpreter startup instead of at interpreter build-time; in all other respects this implementation behaves like normal static replication (i.e., the replication is not specific to the interpreted program, unlike dynamic replication). This was easier to implement, allowed to use more replication configurations (in particular, more replicas) and should produce the same results as normal static replication (except for the copying overhead, and the impact of that was small compared to the benchmark run-times).

We implemented dynamic methods pretty much as described in Section 5.2, with free choice (through command-line flags) of replication, superinstructions, or both, and superinstructions within basic-blocks or across them. By using this machinery with a VM interpreter including static superinstructions we can also explore the combination of static superinstructions (with optimizations across component instructions) and the dynamic methods.

One thing that we have not implemented is eliminating the increments of the VM instruction pointers along with the rest of the instruction dispatch in dynamic superinstructions. However, by using static superinstructions in addition dynamic superinstructions and replication we also reduce these increments (in addition to other optimizations); looking at the results from that, eliminating only the increments probably does not have much effect. It would also conflict with superinstructions across basic blocks.

6.2 Machines

We used an 800MHz Celeron (VIA Apollo Pro chipset, 512MB PC100 SDRAM, Linux-2.4.7, glibc-2.2.2, gcc-2.95.3) for most of the results we present here. The reason for this choice is that the Celeron has a relatively small I-cache (16KB), L2 cache (128KB), and BTB (512 entries), so any negative performance impacts of the code growth from our techniques should become visible on this processor.

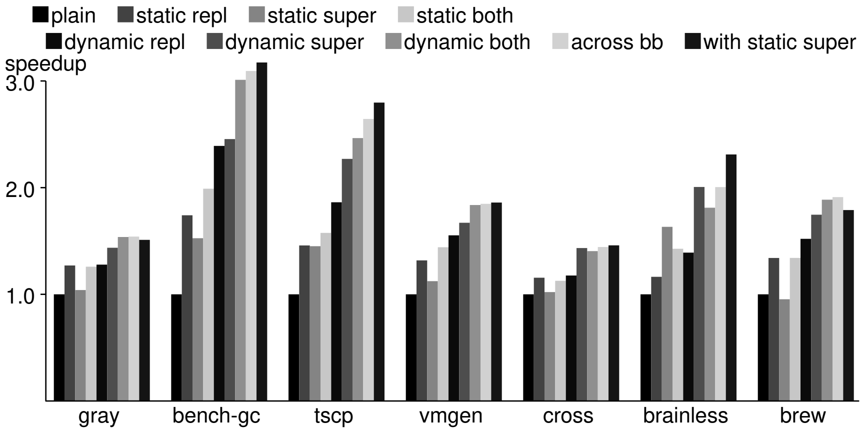


Figure 10: Speedups of various interpreter optimizations on a Celeron-800

For comparison, we also present some results from a 1200MHz Athlon (Thunderbird; VIA KT133 chipset, 192MB PC100 SDRAM, Linux-2.4.0, glibc-2.1.3, gcc-2.95.1). This processor has a larger I-cache (64KB), L2 cache (256KB), and BTB (2048 entries).

Both processors allow measuring a variety of events with performance-monitoring counters, providing additional insights.

6.3 Benchmarks

Figure 9 shows the benchmarks we used for our experiments. The line counts include libraries that are not preloaded in Gforth, but not what would be considered as input files in languages with a hard compile-time/run-time boundary (e.g., the grammar for gray, and the program to be compiled for cross), as far as we could tell the difference.

7. RESULTS

7.1 Interpreter variants

We compared the following variants of Gforth:

plain Threaded code; this is used as the baseline of our comparison (factor 1).

static repl Static replication with 400 replicas and round-robin selection.

static super 400 static superinstructions with greedy selection.

static both 35 unique superinstructions, 365 replicas of instructions and superinstructions (for a total of 400).

dynamic repl Dynamic replication

dynamic super Dynamic superinstructions without replication, limited to basic blocks (very similar to what Piumarta and Riccardi proposed [13]).

dynamic both Dynamic superinstructions, limited to basic blocks, with replication.

across bb Dynamic superinstructions across basic blocks, with replication.

with static super First, combine instructions within a basic block into static superinstructions (with 400 superinstructions) with greedy selection, then form dynamic superinstructions across basic blocks with replication from that. This combines the speed benefits of static superinstructions (optimization across VM instructions) with the benefits of dynamic superinstructions with replication.

We used the most frequently executed VM instructions and sequences from a training run with the *brainless* benchmark for static replication and static superinstructions.

We used 400 additional instructions for the static variants because it is a realistic number for interpreters distributed in source code: it does not cost that much in interpreter compile-time and compile-space, and using more gives rapidly diminishing improvements.

The presented results are for complete benchmark runs, including interpreter startup times, benchmark compilation, and, for the dynamic variants, the time spent in code copying.

7.2 Speedups

Figure 10 shows the speedups these versions achieve over *plain* on various benchmarks.

The dynamic methods fare better than the static methods (exception: on brainless static superinstructions do better than dynamic replication; that is probably because the training program was brainless).

For the static methods, we see that static replication does better than static superinstructions, probably because replication depends less on how well the training run fits the actual run. A combination of replication and superinstructions is usually better, however (see Section 7.5).

For the dynamic methods, superinstructions alone perform better than replication alone. However, the combination performs even better; exceptions: cross and brainless on the Celeron, due to I-cache misses (on the Athlon the combination is better for all benchmarks). Performing both optimizations across basic blocks is always beneficial, and using static superinstructions in addition helps some more (exception: brew, because static superinstructions do not improve the prediction accuracy there and because it executes more native instructions; this is an artifact of the im-

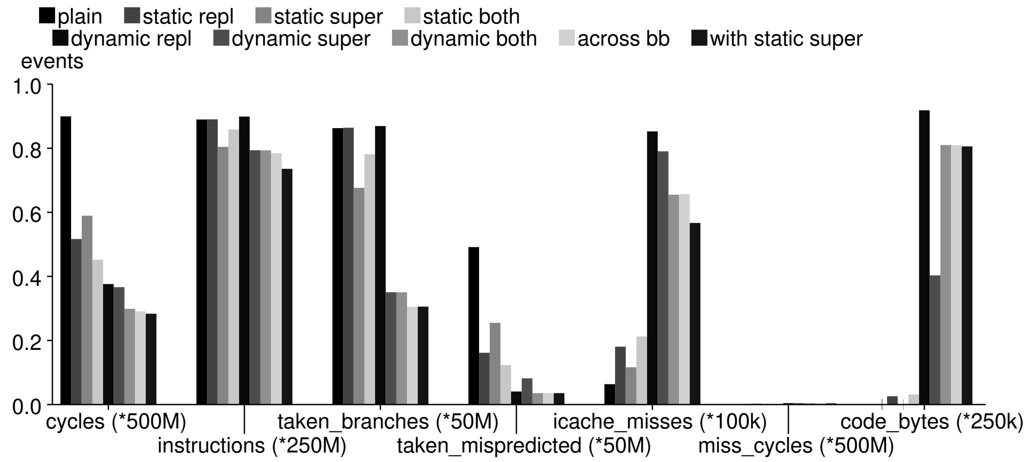


Figure 11: Performance counter results for bench-gc on a Celeron-800

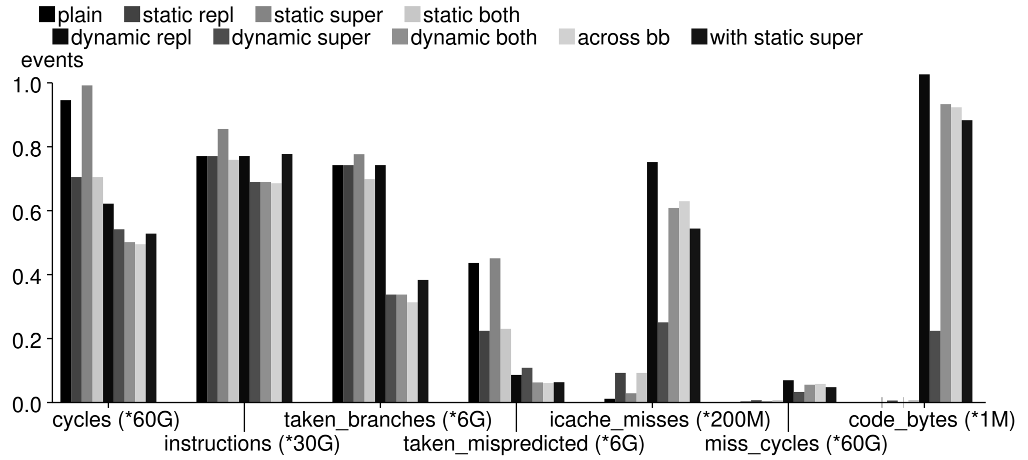


Figure 12: Performance counter results for brew on a Celeron-800

plementation of superinstructions in this version of Gforth and does not transfer to other interpreters or future versions of Gforth).

Overall, the new techniques provide very nice speedups over the techniques usually used in efficient interpreters (up to factor 2.08 for *static both* over *plain*, and factor 3.17 for *with static super* across *plain*), but also across existing techniques that are not yet widely used (factor 1.30 for *static both* over *static super* [8] on bench-gc, and factor 1.29 for *with static super* over *dynamic super* [13]).

7.3 Other metrics

We take a closer look at the reasons for the speedups by looking at various other metrics, using mostly performance monitoring counters:

cycles (tsc) The number of cycles taken for executing the program; this is proportional to the reciprocal of the speedup.

instructions (event C0) Executed (retired) instructions.

taken_branches (event C9) Executed (retired) taken branch instructions.

taken_mispredicted (event CA) Executed (retired) taken branch instructions that are mispredicted. For *plain* most of these mispredictions are mispredictions of the dispatch indirect branches of the interpreter. We use the same scale factor for this event as for taken_branches, so you can directly see how many of the taken branches are mispredicted. We also scale this event such that 1 misprediction corresponds to 10 cycles (the approximate cost of a misprediction on a Celeron or Athlon); this allows you to directly see how much of the time is spent in mispredictions and compare this to, e.g., the time spent in I-cache misses.

icache_misses (event 81) Instruction fetch misses. Note the scale factor for these events; they are much rarer than the others.

miss_cycles (event 86) Cycles during which the instruction fetch is stalled (usually due to I-cache misses); we use the same scale factor for this event as for cycles, so you can directly see how much of the time is spent in I-cache misses, and compare this to, e.g., taken_mispredicted.

code_bytes The size of the code generated at run-time, in bytes. Due to the way we implemented static replication, you see a few KB of code generated even for some static schemes.

Figure 11 shows these metrics for *bench-gc*. In this benchmark nearly all of the executed branches are dispatch branches [7], so the effects of our dispatch optimizations should be most evident there.

Figure 12 shows these metrics for *brew*. This is our largest benchmark, so it may unveil effects from code growth that are not apparent with smaller benchmarks (however, *brainless* and *cross* have a slightly higher proportion of I-cache miss cycles, so the locality characteristics of a program do not necessarily correlate with size).

The first thing to notice is that both the instructions and the taken_branches count are the same for *plain*, *static repl*, and *dynamic repl*. Similarly, they are the same for *dynamic super* and *dynamic both*. The reason is that (after startup, with its negligible copying overhead) these interpreters execute exactly the same sequence of native instructions, only coming from different copies of the code. So the difference in cycles between these interpreters comes from the difference in branch mispredictions, I-cache misses and other, similar effects (however, looking at the data, we believe that other effects only play a negligible role).

Looking at the cycles and taken_mispredicted metrics, we see that mispredictions consume a large part of the time in the *plain* interpreter, and that just eliminating most of these mispredictions by dynamic replication gives a dramatic speedup (factor 2.39 for *bench-gc*). Our simulations show that the remaining mispredicted dispatch branches are due to indirect VM branches (mostly, VM returns), apart from capacity and conflict misses in the BTB.

The static methods do not work that well: they do not reduce the mispredictions as much, because they have to reuse VM instructions.

Dynamic superinstructions without replication have a slightly worse misprediction accuracy than dynamic replication, because superinstructions are reused, but they make up for this by executing fewer instructions, and (for brew) taking fewer miss_cycles.

Looking at the instructions, we see that VM superinstructions do not reduce the number of executed native instructions much. Both static and dynamic superinstructions reduce this by similar amounts (apart from brew); dynamic superinstructions eliminate more dispatch code (see also the effect on taken_branches), whereas static superinstructions allow optimizations between component VM instructions. *Across bb* reduces the instructions a little more, and *with static super* also a little more (exception: brew).

Looking at taken_branches, we get a similar picture as with instructions, except that dynamic superinstructions reduce this metric much more than static superinstructions. Also, *across bb* and *with static super* have the same number of taken_branches (exception: brew), because *with static super* only changes what goes on in a dynamic superinstruction, not how it is formed.

Taken_branches also indicates (and our simulation results confirm) that the length of the average executed superinstruction is quite short for static superinstructions (typically around 1.5 component instructions), but also for dynamic superinstructions (around 3 component instructions). Also, *across bb* does not increase the superinstruction length by much, because in Forth the most frequent reason for basic block boundaries is calls and returns, and *across bb* does not help there; therefore we expect *across bb* to have a greater effect on superinstruction length and on performance in other languages.

7.4 Code growth

A frequent reaction to the proposal for replication is that the resulting code growth will cause so many performance problems that the end result will be slower than the original interpreter; a quick look at the speedups (Fig. 10) should convince everyone that this is not true, even on a CPU with small caches like the Celeron. Still, in this section we take a closer look at the code growth and its effect on various metrics.

In the code_bytes bars of Fig. 12 we see that the dynamic-replication based methods produce about 1MB of native code for brew, with longer superinstructions and static superinstructions reducing the code size a little. In many environments this is quite acceptable for a 30000-line program (e.g., brew also consumes 0.5MB of the Gforth data space containing threaded code and data).

Dynamic superinstructions without replication reuse superinstructions a lot, resulting in a generated code size of only 200KB.

These size differences are also reflected in icache_misses: the static methods have very few misses, *dynamic super* some more, and the replication-based methods even more; this is also reflected in the miss_cycles.

However, the miss_cycles only consume a small part of the total cycles in most cases, and only in a few cases do they overcome the benefit obtained from better prediction accuracy; in particular, on the Celeron *dynamic both* spends 23% of the cycles on misses when running brainless (compared to 7.5% for *dynamic super*), resulting in a slowdown by factor 1.11; however, *dynamic both* is faster for most other benchmarks on the Celeron, and for all benchmarks on the Athlon (factor 1.07 for brainless).

So, unless you have reason to expect to run programs with particularly bad code locality, we recommend using dynamic replication together with dynamic superinstructions for general-purpose machines.

Another way of looking at the issue is to compare the code generated by our replication methods to code generated by a native-code compiler[4]; it will typically be larger than the native code by a small constant factor (the factor may be even < 1 if the native-code compiler uses loop unrolling, inlining, and other code replicating optimizations); for most code I-cache misses are not a big issue, so the code size resulting from replication is usually not a big issue, either.

7.5 Balancing static methods

Figure 13 shows timing results for various combinations of static replication and superinstructions. Each line represents a given number of total additional instructions, varying distributions between replication and superinstructions along the X axis. .

We can see that the performance improves with the total number of additional instructions, but approaches a limit of around 200M cycles.

[4]Unfortunately the native-code Forth compilers we use do not report the size of the generated code, so we cannot present empirical data here.

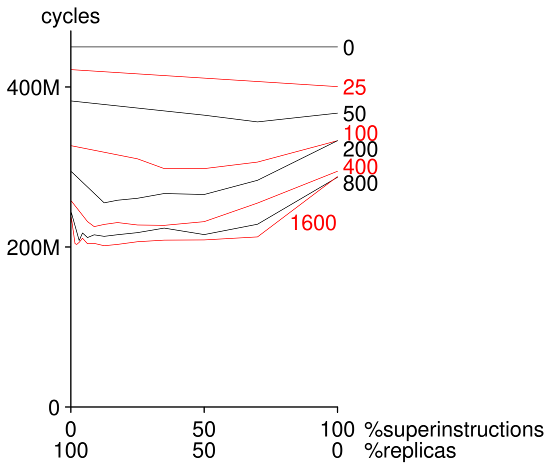


Figure 13: Timing results for *Bench-gc* with static replications and superinstructions on a Celeron-800; the line labels specify the total number of additional VM instructions

	across bb	bigForth	iForth
tscp	2.98	5.13	3.51
brainless	2.49	2.73	
brew	2.17	0.92	

Figure 14: Speedups of *across bb* and two native code compilers over *plain*.

We can also see that a combination of replication and superinstructions gives good results; as long as we are not too close to the extreme points, performance is not very sensitive to the actual distribution between replication and superinstructions.

7.6 Speed comparison with native-code compilers

In this section we look at how far the resulting interpreters are still from relatively simple native-code compilers. The native-code Forth compilers we used are bigForth-2.03 and iForth-1.12. For the data in this section we used Gforth-0.6.1, which gives slightly different speedups from the version used earlier. We also use tscp-0.5. We only ran those benchmarks that we could get to run on the different compilers easily. The benchmarks were run on an Athlon-1200 (Linux-2.4.19, glibc-2.1.3).

You see the results in Fig. 14. Drawing conclusions from such a small sample size (both compilers and benchmarks) is dangerous, but the speed difference between interpreters and native-code compilers appears to be less than many people imagine.

8. RELATED WORK

The accuracy of static conditional branch predictors has been improved with software methods: branch alignment [3] and code replication [12, 18, 17]. The present paper looks at using software methods to improve the accuracy of the BTB, a simple dynamic indirect branch predictor. Our code replication differs from replication for conditional branch prediction in all aspects: our work addresses a *dynamic indirect* branch predictor (the BTB) instead of a *static conditional* branch predictor. Replication for conditional branches works at compile-time and is based on profiling to find correlations between branches to be exploited by replication, and no data is affected; in contrast, our replication changes the representation of the interpreted program at program startup time to decide the replicas to use.

Better indirect branch predictors than BTBs have been proposed in a number of papers [4, 5, 11] and they work well on interpreters [7], but they are not available in hardware yet, and it will probably take a long time before they are universally available, if at all.

There are a number of recent papers on improving interpreter performance [14, 6, 13, 16]. Software pipelining the interpreter [9, 10] is a way to reduce the branch dispatch costs on architectures with delayed indirect branches (or split indirect branches).

Ertl and Gregg [7] investigated the performance of various branch predictors on interpreters, but did not investigate means to improve the prediction accuracy beyond threaded code. In a similar vein, Romer et al. [15] investigated the performance characteristics of several interpreters. They used inefficient interpreters, and thus did not notice that efficient interpreters spend much of their time on dispatch branches.

Papers dealing with superoperators and superinstructions [14, 13, 10, 8] 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 [8]). 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.

9. 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: replicating VM instructions, such that hopefully each replica occurs only once in the working set (speedup up to a factor of 2.39 over an efficient threaded-code interpreter); and combining sequences of VM instructions into superinstructions (speedup up to a factor of 2.45). In combination these techniques achieve an even greater speedup (up to a factor of 3.17).

There are two variants of these optimizations: The static variant creates the replicas and/or superinstructions at interpreter build-time; it produces less speedup (up to a factor of 1.99), but is completely portable. The dynamic variant creates replicas and/or superinstructions at interpreter run-time; it produces very good speedups (up to a factor of 3.09), but requires a little bit of porting work for each new platform. The dynamic techniques can be combined with static superinstructions for even greater speed (up to a factor 3.17).

The speedup of an optimization has to be balanced against the cost of implementing it. In the present case, in addition to giving good speedups, the dynamic methods are relatively easy to implement (a few days of work). Static replication with a few static superinstructions is also pretty easy to implement for a particular interpreter.

The software and data we used for this paper is available at http://www.complang.tuwien.ac.at/anton/interpreter-btb/.

Acknowledgements

We thank the referees for their helpful comments. The performance counter measurements were made using Mikael Pettersson's perfctr package.

10. REFERENCES

[1] J. R. Bell. Threaded code. *Commun. ACM*, 16(6):370–372, 1973.

[2] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice-Hall, 1990.

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

[4] 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.

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

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

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

[8] M. A. Ertl, D. Gregg, A. Krall, and B. Paysan. vmgen — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.

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

[10] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Wiel. A code compression system based on pipelined interpreters. *Software—Practice and Experience*, 29(11):1005–1023, Sept. 1999.

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

[12] A. 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.

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

[14] T. A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Principles of Programming Languages (POPL '95)*, pages 322–332, 1995.

[15] 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)*, pages 150–159, 1996.

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

[17] C. Young, N. Gloy, and M. D. Smith. A comparative analysis of schemes for correlated branch prediction. In 22^{nd} *Annual International Symposium on Computer Architecture*, pages 276–286, 1995.

[18] 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)*, pages 232–241, 1994.