Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters

Kevin Casey
Trinity College Dublin
M. Anton Ertl
TU Wien
and
David Gregg
Trinity College Dublin

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 (BTBs) are the most 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. To show their generality, we have implemented these optimizations in VMs for both Java and Forth. These techniques can eliminate nearly all of the dispatch branch mispredictions, and have other benefits, resulting in speedups by a factor of up to 4.55 over efficient threaded-code interpreters, and speedups by a factor of up to 1.34 over techniques relying on dynamic superinstructions alone.

Categories and Subject Descriptors: D.3 [**Software**]: Programming Languages; D.3.4 [**Programming Languages**]: Processors—*Interpreters*

General Terms: Languages, Performance, Experimentation

Additional Key Words and Phrases: Interpreter, branch target buffer, branch prediction, code replication, superinstruction

Correspondence Address: David Gregg, Department of Computer Science, University of Dublin, Trinity College, Dublin 2, Ireland. `David.Gregg@cs.tcd.ie`

An earlier version of the work described in this paper appeared in the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation [Ertl and Gregg 2003a]. The main difference between the versions is that an implementation and experimental results for a Java VM have been added, whereas the earlier paper contained results only for Gforth. These new results show that the techniques described in this paper are general, and applicable to the JVM. We have also solved a number of JVM specific problems, especially those relating to quick instructions.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 1999 ACM 0164-0925/99/0100-0111 $00.75

1. INTRODUCTION

Implementers of programming languages can choose between a number of different implementation technologies, such as ahead-of-time compilation, just-in-time (JIT) compilation, and interpretation. Each of these has its own advantages and disadvantages in areas such as ease of implementation, compilation speed, and execution speeed, which make each appropriate for different situations.

Interpreters are frequently used to implement virtual machines because they have several practical advantages over native code compilers. If written in a high-level language, interpreters are portable; they can simply be recompiled for a new architecture, whereas a JIT compiler requires considerable porting effort. Interpreters are also dramatically simpler than compilers; they are easy to construct, and easy to debug. Interpreters also require little memory: the interpreter itself is typically much smaller than a JIT compiler, and the interpreted bytecode is usually a fraction of the size of the corresponding executable native code. For this reason, interpreters are commonly found in embedded systems. Furthermore, interpreters avoid the compilation overhead in JIT compilers. For rarely executed code, interpreting may be much faster than JIT compilation. The Hotspot JVMs [Sun-Microsystems 2001] take advantage of this by using a hybrid interpreter/JIT system. Code is initially interpreted, saving the time and space of JIT compilation. Only if a section of code is identified as an execution hot spot is it JIT compiled. Finally, it is easy to provide tools such as debuggers and profilers when using an interpreter because it is easy to insert additional code into an interpreter loop. Providing such tools for native code is much more complex. Thus, interpreters provide a range of attractive features for language implementation. In particular, most scripting languages are implemented using interpreters. The main disadvantage of interpreters is their poor execution speed. Even the fastest interpreters are 5–10 times slower than executable native code.

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 real machine instructions on a RISC architecture). Mispredicted branches are expensive on modern processors (e.g., they cost about 10 cycles on the Pentium III and Athlon and 20–30 cycles on the Pentium 4). As a result, interpreters can spend more than half of their execution time recovering from indirect branch mispredictions [Ertl and Gregg 2003b]. Consequently, improving the indirect branch prediction accuracy has a large effect on interpreter performance. The most widely used branch predictor in current processors is the branch target buffer (BTB). Most current desktop and server processors have a BTB or similar structure, including the Pentium 3 and 4, Athlon and Alpha 21264 processors. BTBs mispredict 50%–63% of the executed indirect branches in threaded-code interpreters and 81%–98% in switch-based interpreters [Ertl and Gregg 2003b].

The rest of this paper is organised as follows. We first introduce the some basic techniques for efficient VM interpreters (Section 2), and explain how indirect branch mispredictions consume a very large proportion of execution time (Section 3). Section 4 proposes two techniques for reducing the incidence of indirect branch mispredictions in VM interpreters. In Section 5 we present static and dynamic variations of each of these technique, and show how they can be efficiently implemented.

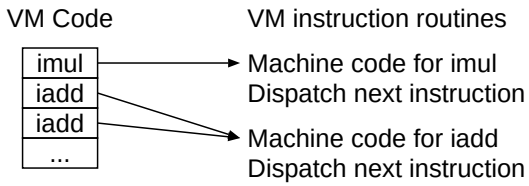
```
typedef enum {
  add /* ... */
} Inst;

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

  Inst *ip = program;
  int *sp;

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

Fig. 1. VM instruction dispatch using `switch`




```
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
```

Fig. 2. Threaded code: VM code representation and instruction dispatch

Section 6 describes our experimental setup for evaluating these techniques and Section 7 describes the results of these experiments. Section 8 compares our approach with existing related work in the area. Finally, we summarise our contributions in Section 9.

2. BACKGROUND

In this section we discuss how efficient interpreters are typically implemented and examine the BTB, a popular branch prediction mechanism. Section 3 will then highlight the significant relationship between interpreters and BTBs.

2.1 Efficient Interpreters

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 is a design which 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.[1] 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 very high proportion of indirect branches in the executed instruction mix. In previous work on a RISC architecture we have measured indirect branches accounting for up to 13% of executed instructions for the Gforth interpreter and 11% for the Ocaml interpreter [Ertl and Gregg 2003b] As we show in Section 7.2.2, on the CISC Intel x86 architecture where more work can be accomplished with fewer real machine instructions, the average for Gforth can be as high as 16.5% of executed instructions.

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 Figure 1), but is not very efficient [Ertl and Gregg 2003b] (see also Section 3).

[1]The term *virtual machine* is used in a number of slightly different ways by various people; the meaning we use is what Smith and Nair [2005] describe as high-level language VMs.

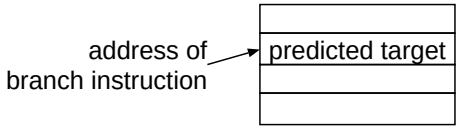


Fig. 3. Branch Target Buffer (BTB)

Threaded code. Represents a VM instruction as the address of the routine that implements the instruction [Bell 1973]. 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, and 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. This is because branches 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 branches is called the misprediction penalty. The misprediction penalty is about 10 cycles on the Pentium III, Athlon, and 21264, about 20 cycles on the earlier Pentium 4 processors with the "Williamette" and "Northwood" cores, and around 30 cycles on the more recent Pentium 4 models with the "Prescott" core.

The most widely used predictor for indirect branches 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 Figure 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 desktop Pentiums, Athlon, Alpha 21264.

Better indirect branch predictors have been proposed [Driesen and Hölzle 1998; 1999; Kalamatianos and Kaeli 1999], and they would improve the prediction accuracy in interpreters substantially [Ertl and Gregg 2003b]. However, they have not been implemented yet in widely available desktop or server hardware. Currently, the only widely available processor which uses a two-level indirect branch predictor

Table I. BTB predictions on a small VM program.

		Switch Dispatch			Threaded Dispatch		
#	VM program:	BTB entry	prediction	actual	BTB entry	prediction	actual
1	*label:* A	*switch*	A	B	*br-A*	GOTO	B
2	B	*switch*	B	A	*br-B*	A	A
3	A	*switch*	A	GOTO	*br-A*	B	GOTO
4	GOTO *label*	*switch*	GOTO	A	*br-GOTO*	A	A

is the Intel Pentium M processor for laptop computers (see Section 8). For the great majority of processors that continue to use a BTB, the software techniques explored in this paper improve the prediction accuracy now, by a similar amount.

3. INTERPRETERS AND BTBS

Ertl and Gregg [2003b] investigated the performance of several virtual machine interpreters on several branch predictors 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? To illustrate this, Table I shows a fragment of VM code representing a loop. It is assumed that the loop has been executed at least once. The rest of the table depicts the behaviour of the BTB as it is accessed after the execution of each VM instruction. These accesses are made upon the completion of each VM instruction when a dispatch to the next instruction must occur. In both the case of switch dispatch and threaded dispatch, the table shows the entry in the BTB which is being accessed, the prediction made by the BTB and finally the actual target at that point. In this way we can see easily when the BTB makes an incorrect prediction, i.e. a branch misprediction.

With switch dispatch, there is only one indirect branch (*br-switch*, the switch branch) and consequently there is only one BTB entry involved. When jumping to the native code for VM instruction A, this BTB entry is updated to point to that native code routine (*br-switch*=A). 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 instruction to point to B (*br-switch*=B), etc. So, with switch dispatch, the BTB always predicts that the current instruction will also be the next one to be executed, which in our isolated example is never correct and is, in general, rarely correct.

Table II. Improving BTB prediction accuracy by replicating VM instructions.

		Threaded Dispatch		
#	VM program:	BTB entry	prediction	actual
1	*label:* A_1	*br-A* $_1$	GOTO	B
2	B	*br-B*	A_2	A_2
3	A_2	*br-A* $_2$	B	GOTO
4	GOTO *label*	*br-GOTO*	A_1	A_1

On the other hand, in threaded code each VM instruction has its own indirect branch and corresponding BTB entry [2] (instruction A has branch *br-A* and a BTB entry for *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 (alternately 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.

To see how replication works, consider Table II which shows a similar VM code fragment to that of Table I. This time however, there are two copies of the VM instruction A namely, A_1 and A_2. Each of these copies has its own dispatch branch and its own entry in the BTB (e.g. *br-*A_1 for A_1 and *br-*A_2 for A_2). Because A_1 is always followed by B, and A_2 always 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).

In special circumstances, depending on the replication method chosen, and the number of replicated instructions, it may be possible to inadvertently *increase* branch mispredictions by adding replicated instructions. An example of this can be seen in Table III. Once more, we use a similar (but slightly longer) VM code

[2](assuming there are no conflict or capacity misses in the BTB)

Table III. Increasing mispredictions through bad static replication.

	Original Code			Modified Code		
#	VM program:	prediction	actual	VM program:	prediction	actual
1	*label:* A	GOTO	B	*label:* A	GOTO	B_1
2	B	A	A	B_1	A	A
3	A	B	B	A	B_1	B_2
4	B	A	A	B_2	A	A
5	A	B	GOTO	A	B_2	GOTO
6	GOTO *label*	A	A	GOTO *label*	A	A

Table IV. Improving BTB prediction accuracy with superinstructions.

		Threaded Code		
#	VM program:	BTB entry	prediction	actual
1	*label:* A	*br-A*	B_A	B_A
2	B_A	*br-B_A*	GOTO	GOTO
3	GOTO *label*	*br-GOTO*	A	A

fragment to that of Table I. In this particular example though, the instruction B will be replaced with two replicas (B_1 and B_2). In the original code of Table III, the execution of two instances (VM instructions 1 and 5) of A caused a branch misprediction, while the middle instance of A (VM instruction 3) has a correctly predicted dispatch. In the modified code of Table III, all instances of A cause mispredictions. The reason for this is that originally the first two instances of A were followed by an instruction B, but now the two instances are followed by different versions of B, causing the BTB to always predict the wrong target for A. This increases the number of mispredictions caused by each iteration of this loop from two to three.

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 [Proebsting 1995; Piumarta and Riccardi 1998; Hoogerbrugge et al. 1999; Ertl et al. 2002]. 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 again the loop in Table II. The indirect branch at the end of VM instruction A will mispredict each time. However, if we can combine the sequence B A into the superinstruction B_A (see Table IV) then the outcome is very different. 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. STATIC VERSUS DYNAMIC APPROACHES

There are two different ways that replication and superinstructions can be used to optimize an interpreter (see Figure 4). The primary difference between these approaches is the point in time when the extra instructions (either replications or superinstructions) are to be introduced, either at build-time or run-time. Each has its own advantages and disadvantages.

5.1 Static Approach

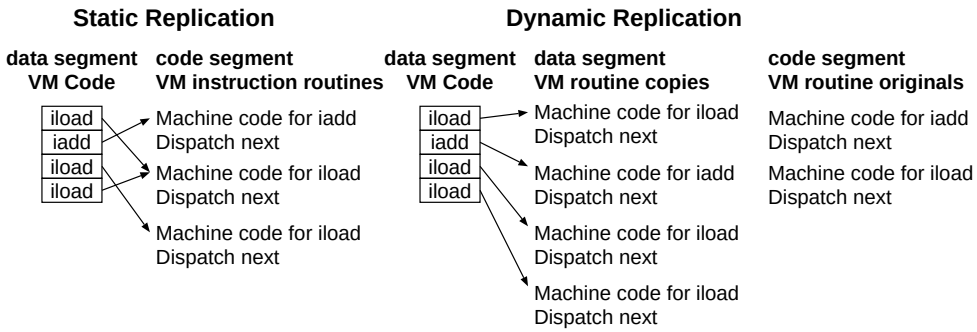


Fig. 4. The static and the dynamic approach to implementing replication

In the static approach, the set of replicas and/or superinstructions is determined at interpreter build-time. The interpreter writer may add additional instructions to the VM manually, or use a macro processor or interpreter generator to create them automatically (e.g., vmgen supports static superinstructions [Ertl et al. 2002]). 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 examined both approaches, and achieved better results for round-robin [Ertl and Gregg 2006], 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 in that region it is less likely to encounter the same replica twice with round-robin selection. E.g., in our example loop we will get perfect branch prediction (Table II) 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.

An important question is how the interpreter front end should choose which superinstructions to apply to a given basic block. With a given set of static superinstructions there may be several different legal replacements of simple instructions with superinstructions. In fact, this problem is similar to dictionary-based compression of text using a static dictionary. The text compression literature [Bell et al. 1990] provides us with two standard approaches. The optimal algorithm uses dynamic programming to find the minimum number of (super)instructions for a given

basic block. A simpler and faster alternative is the greedy (maximum munch) algorithm. We have implemented both in our Java interpreter (for details see our technical report [Casey et al. 2005]) and found that there is almost no difference between the results for greedy and optimal selection, but the optimal algorithm is a little slower. For this reason we use the greedy algorithm in the rest of this paper.

5.2 Dynamic Approach

In the dynamic approach the replicas or superinstructions are created when the VM code is produced at interpreter run-time. However, there must be some way to create new executable code in the VM interpreter at run time. Piumarta and Ricardi [1998] and Rossiand Sivalingam [1996] both independently proposed a strategy of copying the executable code at run-time to implement dynamic superinstructions which allows new executable code to be created in an architecture-neutral fashion, and we use a similar approach. The main advantage is that the replicas and superinstructions can be customized to a particular program. However, implementing dynamic code copying is a little more complex than creating static copies of the code, and the resulting code does not benefit from compiler optimizations that can be applied to static superinstructions.

In a simple, unoptimized interpreter there is a section of executable code to implement each VM instruction in the VM instruction set. With dynamic replication, there is a separate copy of this executable code for every instance of every instruction in the program. This ensures that indirect branches (and, in the absence of capacity and conflict misses, BTB entries) are never shared between different instances of the same VM instruction. A disadvantage of this approach is that it creates a lot of extra executable code but it does not reduce the number of indirect branches to be executed. Instead, an indirect branch must be executed at the end of each VM instruction, but there will normally be only one target of this indirect branch (some instructions, such as VM branches, may have more than one target).

Dynamic replication is simple to implement. Each 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 Figure 4). 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 by virtue of a label that is placed there.

In contrast to replication, dynamic superinstructions greatly reduce the number of dispatches in the program. A superinstruction is created for every basic block in the program, which means that instruction dispatches only occur at the end of a basic block. In addition, the dispatch code between VM instructions within a basic block does not need to be copied, so dynamic superinstructions create less code growth than dynamic replication. Furthermore, if more than one basic block consists of the same sequence of VM instructions, only a single dynamic superinstruction is created [Piumarta and Riccardi 1998]. Thus, there is substantial code reuse, and the code growth is much smaller than that created by dynamic replication.

One weakness of dynamic superinstructions is the sharing of executable code for multiple identical basic blocks, because it makes the dispatch indirect branch at the end of the superinstruction less predictable. This is especially true for basic

blocks that consist of only one or two VM instructions. Dynamic superinstructions with replication solves this problem by having a separate dynamic superinstruction for every basic block in the program. This is actually simpler to implement than dynamic superinstructions alone, because it is no longer necessary to check for multiple identical basic blocks. On the other hand, dynamic superinstructions with replication requires significant code growth, typically almost as much as dynamic replication alone.

A final variation is dynamic superinstructions larger than a basic block. This is motivated by the desire to eliminate as many redundant dispatches as possible. There is no particular reason why dynamic superinstructions must end at a basic block boundary. In many cases the code from the previous basic block can simply fall through to the current one. It is only necessary for a VM instruction dispatch to take place in the case of a taken VM branch. This approach minimizes the number of VM instruction dispatches, and may have a slight advantage in code size compared to dynamic superinstructions with replication. To our knowledge, we are the first to implement such a technique, so a detailed description of the necessary implementation technique follows.

In order to achieve dynamic superinstructions longer than a basic block, one must deal with VM code entry points (VM jumps into the middle of the dynamic superinstruction) and conditional branches out of the dynamic superinstruction. Two changes are required in order to achieve this.

Firstly, it is necessary to keep the increments of the VM instruction pointer even if one does not copy the rest of the dispatch code; as a result, the VM code will be quite similar to the dynamic replication case (whereas there will be only one threaded-code pointer per superinstruction if one eliminates the increments). This allows to continue the superinstruction across VM code entry points. Now, 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.

Secondly the dispatch for the fall-through path of a conditional VM branch can be removed, as long as one introduces an additional dispatch for the branch-taken path. In this way, if the branch is not taken, execution falls through to the next instruction without an intervening (and redundant) branch. The only branch code in such conditional code will be the branch that is executed only if the branch-taken path is to be followed.

As a result of these two optimizations, all dispatches are eliminated, except dispatches for taken VM branches, VM calls and VM returns (see Figure 5).

One problem with the dynamic approach is that it can only copy code that is relocatable; i.e., it cannot copy code if that 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). The architecture and the C compiler determine whether or not a VM instruction is relocatable; so, a general no-copying list [Piumarta and Riccardi 1998] is not sufficient.

Our approach to this problem is to have two versions of the VM interpreter function, one with some gratuitous 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.

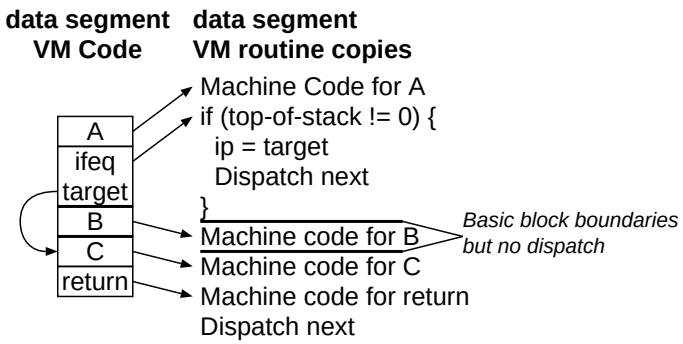


Fig. 5. Superinstructions across basic blocks

The dynamic approach requires a small amount of platform-specific code; on most architectures it needs only code for flushing the I-cache but, for example, 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 the compiler can optimize their code 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 x86 and Alpha architectures. For the JVM, instructions that can throw exceptions are often non-relocatable (due to a relative branch to the throw code outside the code for the VM instruction). In order to make these instructions relocatable in our implementation, we used an indirect branch instead of the relative branch. This approach is also used in the SableVM [Gagnon and Hendren 2001; Gagnon 2003; Gagnon and Hendren 2003] for the same reasons. Since exceptions are thrown very infrequently, this had no measurable adverse effect on performance. Indeed, given that exceptions are computationally costly operations with their dispatch cost only representing a tiny fraction of that cost (a particularly low *dispatch-to-real-work* ratio), the additional cost of the indirect branch approach is even less significant. Once this modification had been made, the most frequently-executed JVM instructions were all relocatable. The implementation of some JVM instructions involved calls to complex external functions that also made the instruction non-relocatable. In such circumstances, the function call was made using a function pointer, rendering the instruction relocatable. The target of this

call is always the same, so it is highly predictable.

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 Forth implementation, 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 Java "quick" instructions

A serious complication in almost any implementation of the JVM is that some VM instructions may have to perform initialization on the first time they are executed. For example, the first time a `getfield` instruction is executed, the field to be fetched must be resolved. The JVM implementation must check that such a field actually exists; it must convert the text name of the field to an offset from the start of the object; it must check the type of field to fetch; and finally it must fetch an item of the appropriate size from the object and place it on the stack. On subsequent executions, no such initializations are necessary, particularly if the type and offset of the field can be cached. The most common solution is to introduce an optimized (or "*quick*") version of the instruction, which does not perform the initializations. On the first time it is executed, the original (or "*quickable*") instruction carries out the initializations, caches any necessary data, and replaces itself with the quick version. Thus, within the JVM implementation, the Java bytecode is modified at run time. We refer to this process as "quickening".

Quickening is relatively straightforward when one is dealing with bytecode or direct threaded code. After running the quickable instruction for the first time, we then can replace the index (in the bytecode case) or the address (in the direct threaded case) of the quickable instruction with that of the corresponding quick instruction. The substitution is not always known prior to the running the quickable instruction for the first time. For example, there are several different quick versions of the `getfield` instruction, each of which loads a different type of field. Thus, when presented with the bytecode for a method, we do not know what some quickable instruction will translate to until it has been executed for the first time.

Quick instructions complicate static replication and superinstructions. Quickable instructions are too rarely executed to replicate. Instead we replicate the quick versions, and choose among the replicas at the time the instruction is quickened.

[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).

Static superinstructions are more complicated for two reasons. First, it would be wasteful to have superinstructions that contain quickable component instructions. These superinstructions would be executed only once, and then be replaced with quick versions. Secondly, for a given quickable instruction, we do not know the corresponding quick instruction until it has actually executed for the first time. We may have a superinstruction that includes a quick `getfield` that loads a byte, but not one that loads an integer, so we only know whether a sequence containing such an instruction can be replaced by a superinstruction after quickening[4]. We solved this problem by adding an additional step to the quickening process, which checks whether the new quick instruction can participate in one or more static superinstructions, and re-parses on that basis. Further details of an earlier version of our static superinstruction implementation were described by Casey *et al* [2003].

Quickening also complicates dynamic replication. We do not create dynamic replicas of quickable VM instructions. Instead, we treat them as non-relocatable instructions and jump to the single copy in the base interpreter. In fact, most of the quickable instructions are indeed unrelocatable, because they perform quite complex operations. However, we also leave space in the copied executable code for the quick version of the VM instruction. During quickening, the quick code is patched into this space.

In a small number of cases, the executable code for different quick versions of a single VM instruction has different lengths, but this does not matter for dynamic replication. The executable code for each VM instruction ends with a dispatch, so we simply jump over the gaps.

Our approach with dynamic superinstructions is similar to the one we use with dynamic replication. When we find a quickable instruction in a sequence for which we are creating dynamic superinstruction, we leave an appropriately sized gap. Into the first memory locations of that gap we insert a copy of the executable code to dispatch a VM instruction. This code dispatches to the quickable version of the instruction. The quickening process replaces this dispatch code with the quick version of the executable code, entirely filling the gap.

Quick instructions present some difficulties when parsing the sequences for superinstructions. With standard static superinstructions (ie without any form of dynamic code copying), we take the approach of re-parsing the method each time an instruction was quickened. This is a relatively cheap option with respect to runtime performance. When mixing static superinstructions with dynamic code copying, we only parse a sequence for superinstructions when its count of quickable VM instructions had reached zero (and in this case to also introduce copies of static superinstructions into dynamic superinstructions).

[4]A simpler approach is used by the SableVM [Gagnon and Hendren 2003; Gagnon 2003] as part of their *preparation sequences* (see Section 8). Specialised versions of quickable instructions are introduced into the code by, for example, resolving the types of the field in `getfield` instructions at the time that the bytecode is translated to threaded code. This allow the length of the gap to be computed ahead of time, so no spurious nops need be introduced. This technique is not specific to preparation sequences and could be incorporated into our method for creating dynamic superinstructions to simplify the implementation.

Table V. Comparison of running time of our base Java interpreter with various JVMs on the SPECjvm98 benchmark programs.

Benchmark	Our Base interpreter	Hotspot interpreter	Kaffe interpreter	Hotspot mixed-mode	Kaffe JIT
javac	30.78	25.68	256.49	6.03	17.52
jack	17.77	17.60	126.33	4.19	15.75
mpeg	81.16	75.69	644.63	5.36	10.79
jess	27.13	19.29	247.02	2.75	18.02
db	59.70	46.47	397.11	13.67	21.79
compress	93.66	82.76	1186.74	7.05	7.19
mtrt	28.31	27.80	338.38	1.95	13.10

6. EXPERIMENTAL SETUP

We have conducted experiments using a simulator as well as experiments using an implementation of these techniques. We used a simulator (for GForth only) 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.

The effect on indirect branch mispredictions is similar in the simulated and real implementations (although simplifying assumptions in our simulation environment caused us to estimate a slightly larger number of conflict misses than we measured on a similarly sized real BTB). Therefore, 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. An early version of this paper containing the simulator results is available as a technical report [Ertl and Gregg 2006].

6.1 Implementation

We implemented the techniques described in Section 4 in Gforth, a product-quality Forth interpreter and in CVM, an implementation of the Java 2 Micro Edition (J2ME). For Gforth, we implemented static superinstructions using vmgen [Ertl et al. 2002]; 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 us to use more replication configurations (in particular, more replicas) and produced 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). For our Java interpreter, we implemented static replication and superinstructions using Tiger [Casey et al. 2005], a Java implementation and extension of vmgen. In the JVM case, static replication was implemented at interpreter build time, because Tiger contains extensive support for compile-time static replication.

We implemented dynamic methods 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)

Table VI. Benchmark programs used in Gforth

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*

Table VII. SPECjvm98 Java benchmark programs

Program	*Description*
_201_*compress*	*modified Lempel − Ziv compression*
_202_*jess*	*Java Expert Shell System*
_209_*db*	*small database program*
_213_*javac*	*compiles* 225,000 *lines of code*
_222_*mpegaudio*	*an MPEG Layer − 3 audio stream decoder*
_227_*mtrt*	*multithreaded ray − tracing program*
_228_*jack*	*a parser generator with lexical analysis*

and the dynamic methods.

We do not eliminate 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 to 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

For the Gforth experiments we used an 800MHz Intel Celeron (VIA Apollo Pro chipset, 512MB PC100 SDRAM, Linux-2.4.7, glibc-2.2.2, gcc-2.95.3[5]). 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. This Celeron processor is based on the Pentium 3 processor, unlike some more recent processors sold under the Intel Celeron brand which are based on the Pentium 4.

For comparison, we also present some results from a 2.26GHz Pentium 4 (Northwood Core, 1GB DDR RAM, Linux-2.6.13.2, glibc-2.2.2, gcc-2.95.4). This processor has a trace cache of 12K micro-ops, an L2 cache of 512KB, and a BTB of 4096 entries.

The modified JVM used a 3GHz Pentium 4 (Northwood core, Intel 865PE chipset, 512 MB DDR 333, Linux-2.6.8, glibc-2.3.3, gcc 2.95) which, as above, has a trace cache of 12K micro-ops, an L2 cache of 512KB, and a BTB of 4096 entries. Although the P4 was capable of Hyper-threading (the name used by Intel for its

[5]Implementing dynamic superinstructions is more challenging in GCC 3.x due to certain optimizations that re-order or re-arrange executable code. Full details of these issues can be found in a bug report submitted to GCC. See bug number 15242 at http://gcc.gnu.org/bugzilla/. These problems have been fixed in GCC 4.

implementation of simultaneous multithreading), this was disabled in the BIOS to prevent any difficulties with the performance counters.

All three processors allow measuring a variety of events with performance monitoring counters, providing additional insights into important issues.

6.3 Benchmarks

Table VI shows the benchmarks we used for our experiments with Gforth. 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), in so far as we were able to differentiate between the two.

The benchmarks we chose to examine the effect of various optimisations on our Java interpreter are from the SPECjvm98 suite, issued by the System Performance Evaluation Corporation (SPEC). The programs used from the suite are listed in Table VII.

7. RESULTS

In order to evaluate the various optimizations discussed in this paper, we present data from both a Gforth and a Java interpreter in a variety of configurations. We also present hardware counter results for both interpreters which assists in determining the effect of these optimizations, particularly in relation to branch misprediction and instruction-cache misses.

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 Ricardi [1998] proposed).

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. For the JVM implementation, we experimented with a wide variety of

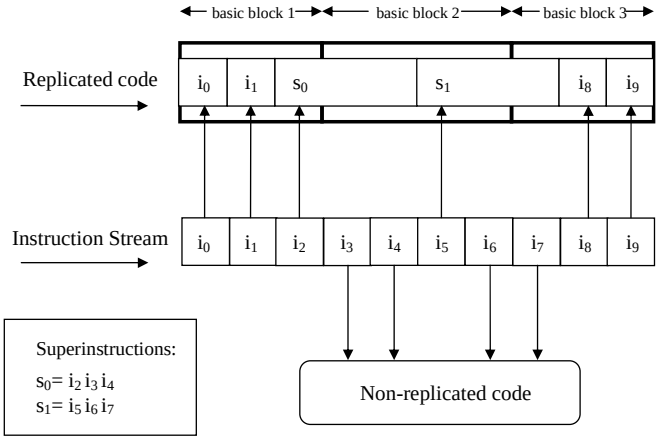


Fig. 6. Adding static superinstructions across basic-blocks to dynamically replicated code

superinstruction selection algorithms. The scheme we finally used was to select static superinstructions and static replicas separately for each benchmark. For example, for compress, we made our selection by profiling all SPECjvm98 benchmark programs *except* compress. We selected the most frequently statically appearing sequences of VM instructions across all programs examined. For superinstructions, we gave shorter sequences a higher weighting because they are more likely to appear in other programs (for details see our technical report [Casey et al. 2005]).

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 variants of our JVM we examined are identical to those listed above with the exception of *static both* for reasons that will become apparent in Section 7.5. Once more, and for the same reasons as before, we use 400 additional instructions for the static variants. Instead, for the JVM result, we present an additional variant *with static across bb*, which is similar to *with static super* except that static superinstructions were permitted to cross basic blocks. This meant we now had to deal with the issue of side-entries, i.e. control flow into the middle of a static superinstruction. The approach taken here was that, when a side-entry occurred, we executed non-replicated instructions until the end of the superinstruction, and then proceeded to execute replicated code once again. The tradeoff with this approach is that on one hand we can include many more static superinstructions in our dynamically replicated code, but on the other hand, any time a side entry occurs into a dynamically replicated static superinstruction, we revert to executing non-replicated code (see Figure 6) for the duration of that dynamically replicated static superinstruction.

7.2 Speedups

In this section we present improvements in running time caused by the various optimization techniques. The presented results are for complete benchmark runs, including interpreter startup times, and the time required to apply replication and superinstructions.

7.2.1 *Gforth speedups.* Figures 7 and 8 show the speedups these versions of Gforth achieve over *plain* on various benchmarks. For the dynamic methods in Gforth, superinstructions alone perform better than replication alone. However, the combination performs even better, except for *cross* and *brainless* on the Celeron, due to I-cache misses (on the Pentium 4 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 implementation of superinstructions in this version of Gforth and does not transfer to other interpreters or future versions of Gforth).

For Gforth, 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).

Overall, the new techniques provide very nice speedups on the Pentium over the techniques usually used in efficient interpreters (up to factor 2.35 for *static both* over *plain*, and factor 4.55 for *with static super* over *plain*), but also over existing techniques that are not yet widely used (factor 1.39 for *static both* over *static super* [Ertl et al. 2002] on bench-gc, and factor 1.34 for *with static super* over *dynamic super* [Piumarta and Riccardi 1998]).

7.2.2 *JVM speedups.* For our JVM (Figure 9), the results are similar, with dynamic methods usually outperforming static methods. However, when executing *mtrt* with dynamic methods, the number of I-cache misses is around ten times higher than with static methods. As a result, for *mtrt*, static superinstructions clearly outperform any dynamic method. In this benchmark, the dynamic method that most closely matches static superinstructions is dynamic superinstructions. This is what is expected since, of all the dynamic methods, dynamic superinstructions is the one which gives the least increase in I-cache misses.

When examining the dynamic methods in our JVM, we find similar results, in that dynamic superinstructions outperform dynamic replication. However, when allowing dynamic replication to extend across basic blocks, the tables are turned and dynamic replication gives marginally better performance. This is due to the reduction in branch-mispredictions (see Figures 12 and 13) and occurs despite an increase in the I-cache miss count.

Overall, the speedups are not as spectacular as for Gforth, but are still appreciable. The best speedup we saw across all benchmarks was 2.76 over *plain* for *with*

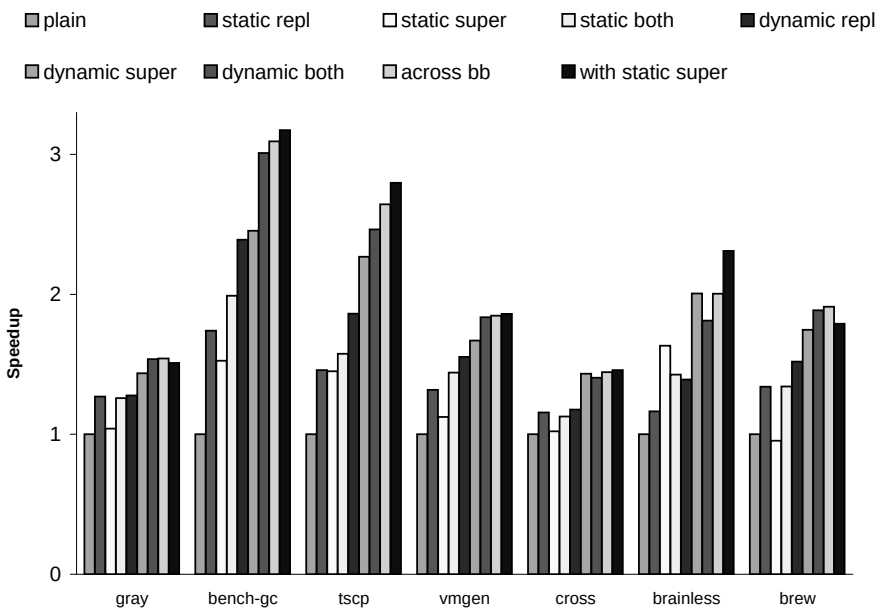


Fig. 7. Speedups of various Gforth interpreter optimizations on a Celeron-800

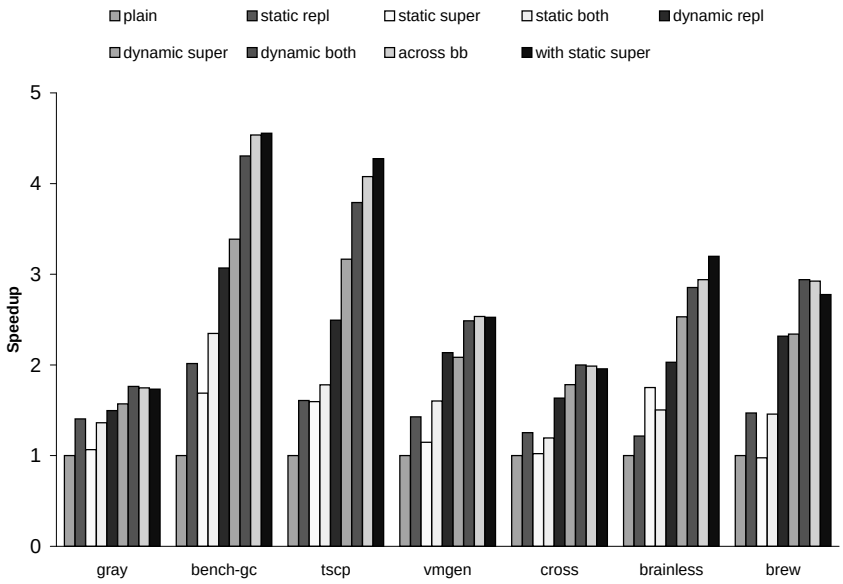


Fig. 8. Speedups of various Gforth interpreter optimizations on a Pentium 4

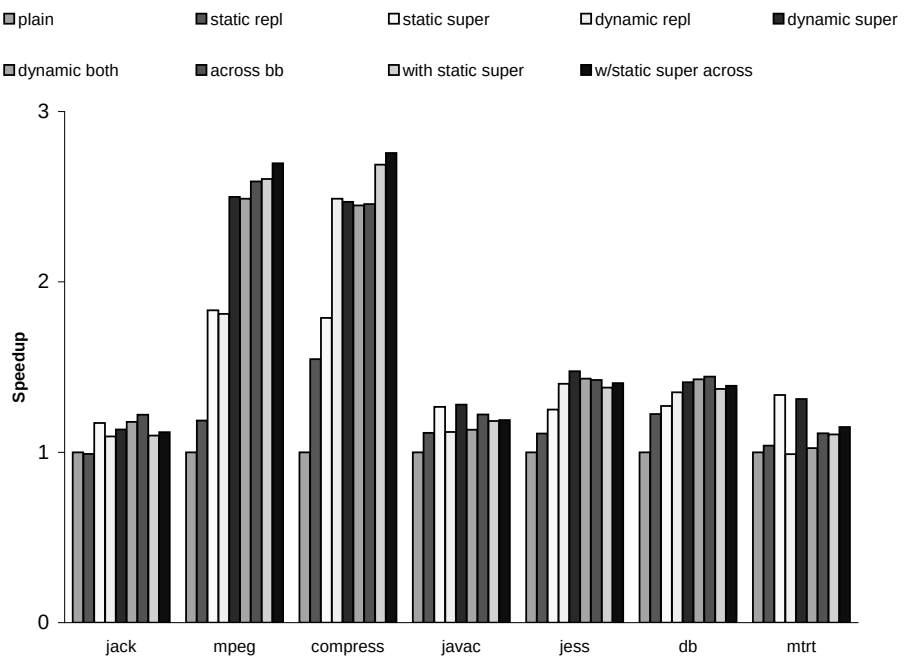


Fig. 9. Speedups of various Java interpreter optimizations on a Pentium 4

static across bb on *compress*. Of the static methods we examined, *static super* gave the best performance with a speedup of 1.83 over *plain* on *mpeg*.

There are three main reasons for the disparity in performance improvements between the Gforth and Java virtual machines. The first is that Gforth VM instructions tend to be simpler instructions. Consequently the dispatch-to-real-work ratio of Gforth programs is quite high. On the other hand, Java VM instructions tend to be more complex instructions. As a result, JVM programs tend to have a fewer dispatches than Gforth programs for the same amount of work performed. A second reason for the discrepancy between performance improvements has to do with the fact that our optimizations are focussed on the interpreter portion of the Java and Gforth virtual machines. However, while the Gforth VM spends almost all of its time in the interpreter, the Java VM spends a considerable portion of its time outside the interpreter, executing such tasks as garbage collection and bytecode verification. A third reason is that in the Gforth VM, the topmost stack item is cached in a register, reducing memory traffic in the interpreter. No corresponding optimization is implemented in our Java VM implementation. The result of these differences is that we have found that, across the seven Gforth benchmarks used in this paper, an average of 16.54% of all executed (retired) real machine instructions are indirect branches (on a Northwood-core P4). In contrast, across the seven SPECjvm98 benchmarks (again on a Northwood-core P4), only 6.08% of all executed real machine instructions are indirect branches. Thus, any of our optimizations which reduce the number of branch mispredictions will have a greater effect in Gforth programs.

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. Executed (retired) instructions. The Pentium 4 splits a small number of complex Intel x86 instruction into simpler RISC-like micro-operations at the time that they are loaded into the instruction cache. The performance counters on this processor count micro-operations rather than x86 instructions, so these are the numbers we present. However, for several of the benchmarks we have compared the number of retired Pentium 4 micro-operations with the number of retired x86 instructions when the same program is run on a Pentium 3, and found that the difference between the two is typically less than 1%.

indirect_branches. Executed (retired) indirect branches are measured directly on the Pentium 4.

mispredicted_indirect. Executed (retired) indirect branch instructions that are mispredicted. We use the same scale factor for this event as for indirect_branches, so one can directly see how many of the taken branches are mispredicted. We also scale this event such that 1 misprediction corresponds to 20 cycles (the approximate cost of a misprediction on a Northwood-core P4); this allows one 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. Instruction fetch misses. Note the scale factor for these events; they are much rarer than the others.

miss_cycles. On the Pentium 4 there is no single counter that measures the cost in cycles of an icache miss. The main reason is that the Pentium 4's L1 icache is a trace cache. A trace cache miss does not simply involve waiting for the next level of memory to return the next instruction. The fetched instructions must be loaded into the trace cache, a process that includes translating them from x86 instructions to RISC micro-operations. Although there are a number of performance counters relating to the trace cache, Intel has not released enough information to be able to combine these counters into a single result. Zhou and Ross [2004] have estimated that a Pentium 4 trace cache miss incurs a penalty of at least 27 cycles. In the absence of a better measure, we have simply multiplied the number of icache misses by 27, in order to estimate the number of cycles lost to icache misses.

code_bytes. The size of the code generated at run-time, in bytes. Due to the way we implemented static replication, one sees a few KB of code generated even for some static schemes in the Gforth measurements (but not in the JVM where we implemented true static replication).

Figure 10 show these metrics for Gforth running *bench-gc*, and Figure 11 shows the corresponding metrics for Gforth running *brew*. The latter is our largest Forth 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 indirect_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 for the Forth benchmarks, 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 3.07 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 [Ertl and Gregg 2006].

The static methods do not work as well: they do not reduce the mispredictions as much, because they have to reuse VM instructions. Nonetheless, the speedup is significant, especially given that static techniques are less complicated to implement.

Dynamic superinstructions without replication have a slightly worse prediction accuracy than dynamic replication, because superinstructions are reused, but they make up for this by executing fewer instructions, and (for brew) by causing 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 indirect_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 indirect_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 indirect_branches (exception: brew), because *with static super* only changes what goes on in a dynamic superinstruction, not how it is formed.

Indirect_branches also indicates (and our simulation results [Ertl and Gregg 2006] 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.

Figures 12 and 13 show the corresponding metrics for our Java interpreter running the mpegaudio and compress benchmarks respectively. As with the Gforth results, the instructions and indirect branches counts are the same for *plain*, *static repl*, and *dynamic repl*. However the execution time is much lower for *static repl*, and *dynamic*

repl because of a large reduction in the number of indirect branch mispredictions. In the case of compress, *dynamic repl* is almost a factor of three times faster than *plain*, and this reduction is entirely attributable to reductions in indirect branch mispredictions.

Static replication is much less successful in our Java implementation than for Gforth. There are two main reasons for this. First, replicating instructions in our JVM does not make all indirect branches more predictable. When we experimented with adding small number of replicated VM instructions, we discovered that they actually reduce the overall indirect branch misprediction rate [Casey et al. 2005]. The reason is that in our implementation, a couple of VM instructions (especially local variable load VM instructions) account for a very large proportion of indirect branch targets. If those VM instructions are replicated, the indirect branches that jump to them become more polymorphic (i.e. they have a greater number of common targets). With larger numbers of indirect branches, the overall effect is positive — but not as positive as for Gforth where even small numbers of replicas improve performance. The second reason for the poor performance of static replication is that it causes very large numbers of icache misses. As more replicas are added the potential for icache thrashing arises.

In contrast, static superinstructions give much better results for our JVM implementation than for Gforth. There are two main reasons for this contrast. First, basic blocks are longer in the Java benchmarks than in the Forth ones. The normal programming style in Forth is to write large numbers of very short functions (or *words*), so basic blocks are broken by large numbers of calls and returns. Secondly, we put a great deal more effort into selecting suitable superinstructions for the Java implementation. We evaluated a wide variety of strategies [Casey et al. 2005]. We discovered the most practical approach was to examine a variety of other Java programs, and select sequences of VM instructions that appear statically frequently, while favoring shorter sequences over longer ones.

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. An examination of the speedups (Figures 7, 8 and 9) should convince everyone that this is not true, even on a CPU with small caches like the Celeron (see Figure 7). 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 Figure 11 we see that the dynamic-replication based methods produce about 1MB of native code for Gforth running 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).

The code_bytes bars of Figures 12 and 13 show that the code growth from dynamic replication-based methods on our JVM is much larger. This is because these are both large programs, and code is also replicated for the extensive Java class library.

Dynamic superinstructions without replication reuse superinstructions a lot, resulting in a generated code size of only 200KB for Gforth and 500k-800K in our

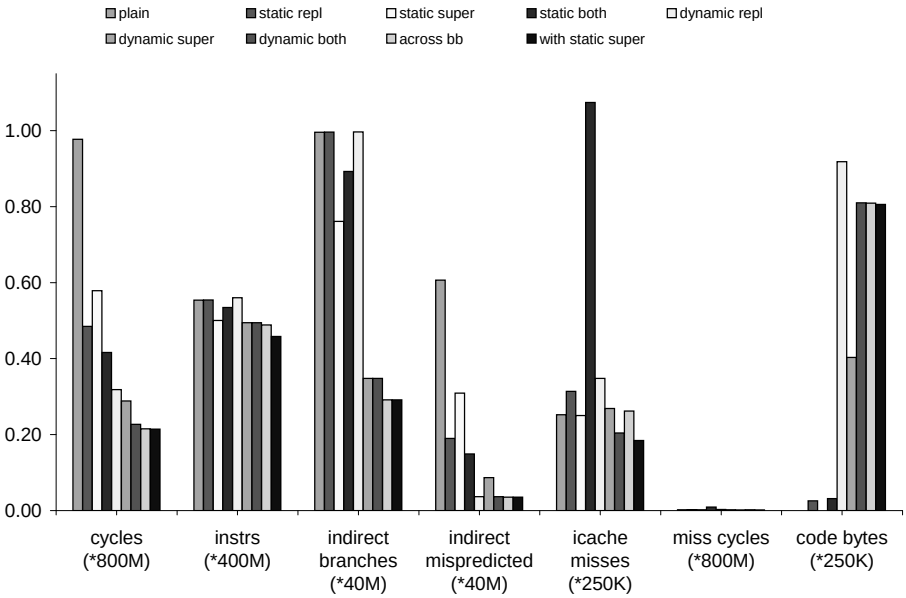


Fig. 10. Performance counter results for *bench-gc* (Gforth) on a Pentium 4

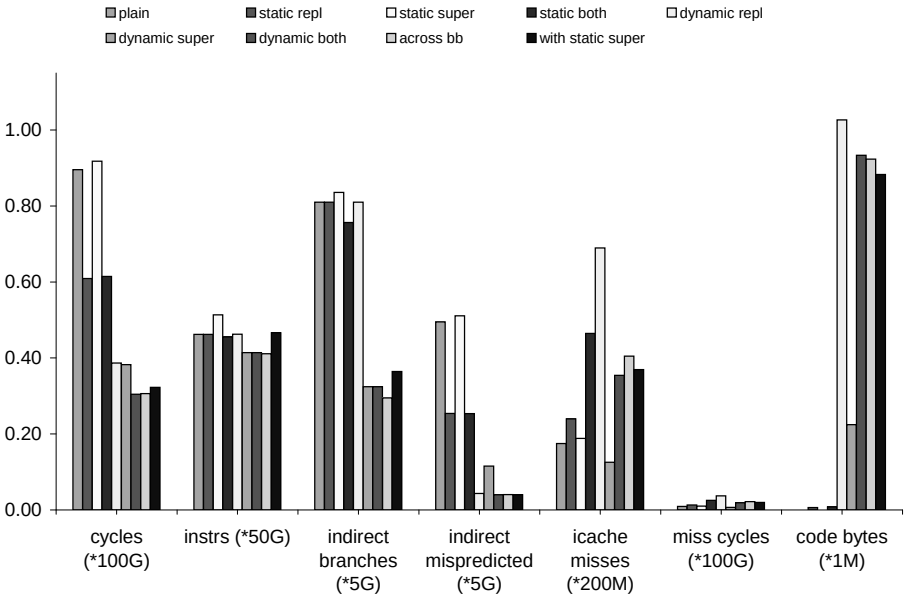


Fig. 11. Performance counter results for *brew* (Gforth) on a Pentium 4

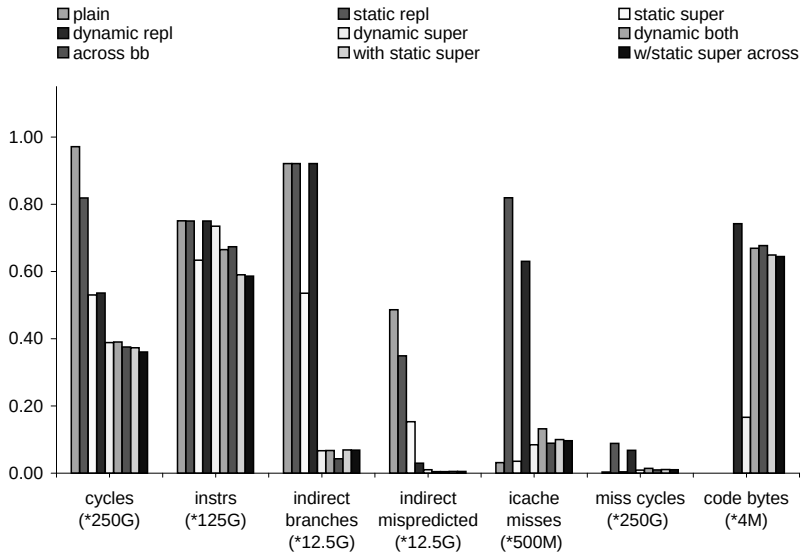


Fig. 12. Performance counter results for *mpegaudio* (Java) on a Pentium 4

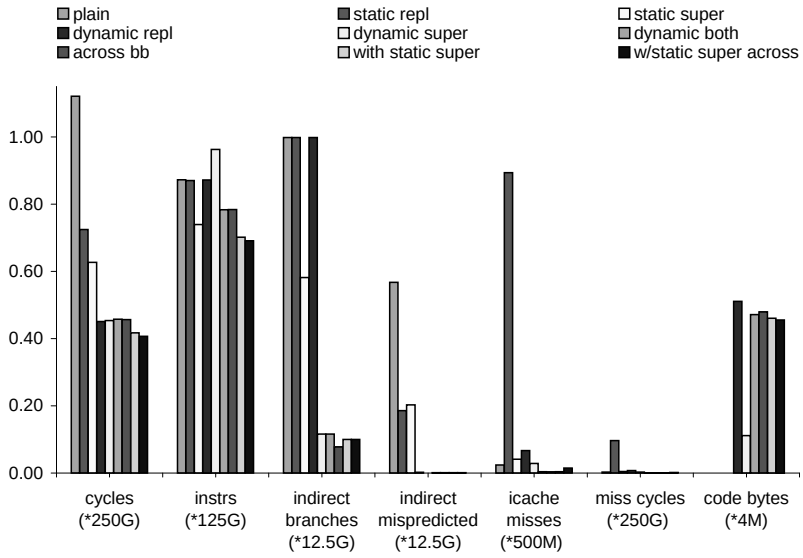


Fig. 13. Performance counter results for *compress* (Java) on a Pentium 4

JVM.

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 consume only a small part of the total cycles in most cases, and only in a few cases do they overcome the benefit obtained from better

Table VIII. Peak dynamic memory requirements (Mb) on various benchmarks

benchmark	*Hotspot (mixed mode)*	*dynamic super*	*across bb*	*w/static across bb*
jack	2.53	0.50	3.08	2.91
mpeg	0.32	0.67	2.71	2.58
compress	0.34	0.45	1.92	1.82
javac	2.63	0.80	4.42	4.22
jess	1.14	0.56	2.76	2.63
db	0.32	0.45	1.99	1.89
mtrt	0.74	0.51	2.40	2.29

prediction accuracy. In particular, hardware counter results from benchmarks on the Celeron (not shown) demonstrated that *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 Pentium 4 (factor of 1.13 speedup for brainless).

So, unless one has 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[6]; it will typically be larger than the native code by a small constant factor (the factor may be even less than one 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.

In order to estimate the additional memory required by a JIT compiler, we used the `memusage` command (from the glibc-utils package) to examine peak heap size in Hotspot's interpreter mode. We then ran the same VM in mixed mode and measured the increase in peak heap size compared to interpreter mode. This gives us an estimate of the additional memory requirements of the mixed-mode over the interpreter mode. These results are presented in Table VIII. Although the Hotspot results should be treated with some caution, due to the nature of estimation, it does appear that at the very least, dynamic super appears to be competitive with the Hotspot VM. Bearing in mind that code reuse in dynamic super is almost certainly higher than that of the Hotspot VM (in mixed mode), this result is expected. At the same time, the Hotspot VM only invokes the JIT on more commonly used methods, so it is no real surprise that *across bb* and *with static super across bb* have much higher memory requirements, since they create (or more precisely replicate) executable code for all methods.

7.5 Balancing static methods

Figure 14 shows timing results for various combinations of static replication and superinstructions in Gforth. Each line represents a given number of total additional instructions, varying distributions between replication and superinstructions along the X-axis.

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

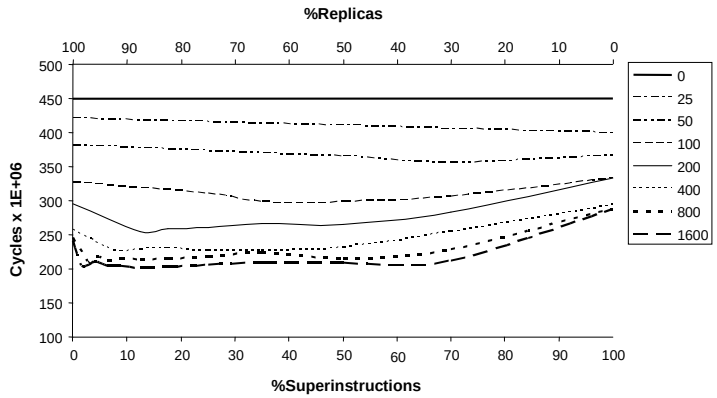


Fig. 14. Timing results for *Bench-gc* (Gforth) with static replications and superinstructions on a Celeron-800; the line labels specify the total number of additional VM instructions

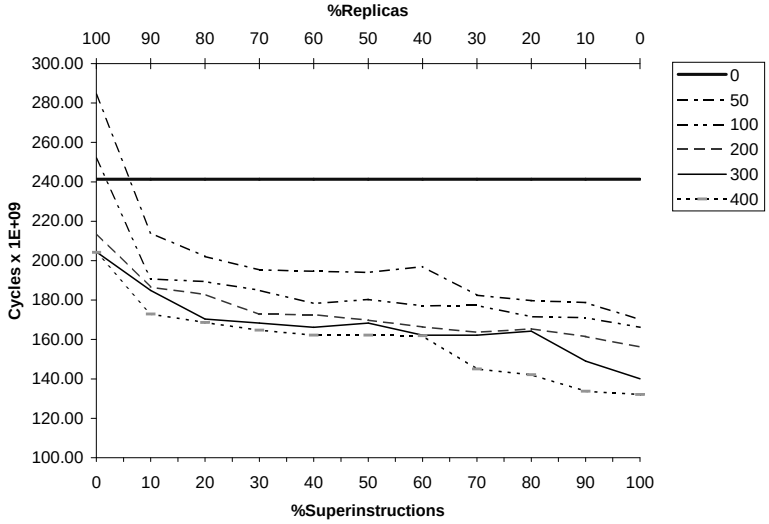


Fig. 15. Timing results for *mpegaudio* (Java) with static replications and superinstructions on a Pentium 4; the line labels specify the total number of additional VM instructions

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

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.

In our Java interpreter, as with Gforth, we attempted to find an optimal mix of static replications and static superinstructions. The results were quite different, however. As seen in Figure 15, there appears to be virtually no benefit in adding replications at the expense of superinstructions. As we noted in the section on static replication, small numbers of static replicated instructions can make performance worse. We see this most clearly when only 50 replications are added (with no static

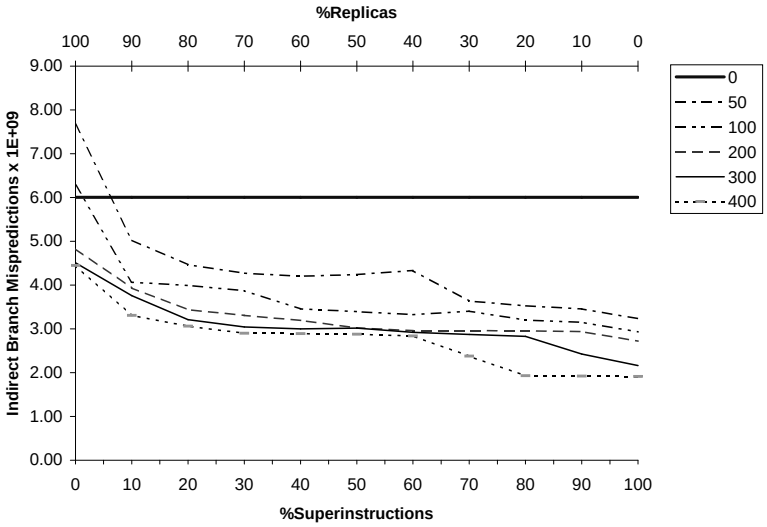


Fig. 16. Indirect Branch Misprediction results for *mpegaudio* (Java) with static replications and superinstructions on a Pentium 4; the line labels specify the total number of additional VM instructions

superinstructions). At this point, we can see from Figure 16 that the number of branch mispredictions actually *increases*. The same effect is observed when only 100 replicated static instructions are added, but this time the effect is not so notable. We examined the branch prediction and branch misprediction hardware counters and at 400 replications (with no superinstructions) the misprediction rate is approximately 27%. In contrast, when using 400 superinstructions, the misprediction rate increases to 30%. However, the superinstruction approach requires only 60% of the executed indirect branches that the replication approach needs, and as a result has only 66% of the number of branch mispredictions.

7.6 Speed comparison with native-code compilers

In this section we look at how far the resulting interpreters are still from 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). The results for Forth are in Table IX. Drawing conclusions from such a small sample size (both compilers and benchmarks) is dangerous, but the speed difference between interpreters and relatively simple native-code compilers appears to be less than many people imagine.

Table X shows corresponding results for various JVM implementations, which are run on the Pentium 4 based system described in Section 6.2. We compare the speedups over plain using *with static across bb*, Kaffe 1.1.4 with the JIT3 engine, and Sun Microsystem's Hotspot client JVM 1.4.2-04-b05 in both interpreter and mixed-mode. Once again the difference between our interpreter and the native code

Table IX. Gforth speedups of *across bb* and two native code compilers over *plain*.

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

Table X. JVM speedups of *w/static across bb*, two native code compilers and an optimised interpreter over *plain*.

	w/static across bb	*kaffe JIT*	*Hotspot (interpreter)*	*Hotspot (mixed mode)*
jack	1.12	1.13	1.01	4.24
mpeg	2.70	7.52	1.07	15.14
compress	2.76	13.02	1.13	13.28
javac	1.19	1.76	1.20	5.11
jess	1.41	1.51	1.41	9.87
db	1.39	2.74	1.28	4.37
mtrt	1.15	2.16	1.02	14.52
average	1.67	4.26	1.16	9.50

compilers is not the orders of magnitude one might expect. Comparing the results of *with static across bb* to those speedups obtained from Hotspot in interpreter mode shows the utility of these methods in optimising interpreters for better performance. The outperforming of Hotspot in interpreter mode is a significant achievement since it has a much faster run time system than CVM. The Hotspot interpreter is also faster than our base interpreter. It is a dynamically-generated, highly-tuned assembly language interpreter, and is able to execute bytecodes more quickly than our portable interpreter written in C.

8. RELATED WORK

An earlier version of the work described in this paper appeared as [Ertl and Gregg 2003a]. The main difference is that the previous version presented only Gforth results. In the current version we have confirmed that the techniques are also useful for other virtual machines, such as the JVM. We have also solved a number of JVM specific problems, especially those relating to quick instructions.

The earlier version of this research inspired some other work in this area. *Subroutine threading* [Berndl et al. 2005] has been proposed as a way of avoiding the cost of indirect branches in VM implementations. Each VM instruction is implemented with a C function. Instead of interpreting bytecode or threaded code, a very simple just-in-time compiler generates executable code for a sequence of calls to these functions. This eliminates indirect branches completely from the dispatch of VM instructions, at the cost of some loss in simplicity and portability.

The accuracy of static conditional branch predictors has been improved with software methods: branch alignment [Calder and Grunwald 1994] and code replication [Krall 1994; Young and Smith 1994; Young et al. 1995]. 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.

There are a number of recent papers on improving interpreter performance [Proebsting 1995; Ertl 1995; Piumarta and Riccardi 1998; Santos Costa 1999]. Software pipelining the interpreter [Hoogerbrugge and Augusteijn 2000; Hoogerbrugge et al. 1999] is a way to reduce the branch dispatch costs on architectures with delayed indirect branches (or split indirect branches).

Romer et al. [1996] investigated the performance characteristics of several interpreters. However, they used inefficient interpreters, and thus did not notice that efficient interpreters spend much of their time on dispatch branches. In contrast, Ertl and Gregg [2003b] investigated the performance efficient interpreters, and found that their running time was often dominated by indirect branch mispredictions. They simulated the performance of various branch predictors on interpreters, but did not investigate means to improve the prediction accuracy beyond threaded code.

In addition to establishing the poor prediction rates of interpreter branches on BTBs, Ertl and Gregg also showed that a processor with a two-level indirect branch predictor can correctly predict most indirect branches in VM interpreters [Ertl and Gregg 2003b]. Hardware two-level branch predictors combine the outcome of the most recently executed indirect branches with the address of the indirect branch, to form an index into a table of target addresses [Driesen and Hölzle 1998; 1999; Kalamatianos and Kaeli 1999]. The first commercially available processor with a two-level predictor is the Intel Pentium M [Gochman et al. 2003] for laptop computers. Details of the predictor are sketchy, but preliminary experiments show that it does indeed correctly predict most indirect branches. At the time of writing, BTBs continue to be the most widely used mechanism for predicting indirect branches, particularly for desktop and server machines. It will probably take a long time before two-level predictors are universally available.

More recently, a BTB which has been optimized just for running Java VMs has been proposed [Li et al. 2005] which combines elements of a two-level predictor and a standard BTB, and achieves similar prediction accuracies to a two-level predictor. However, this hardware predictor is intended only for accelerating VMs, and it is perhaps more likely that general-purpose two-level predictors will appear in hardware before predictors aimed specifically at VMs.

Kaeli and Emma [1994; 1997], describe a *case block table*, a history-based branch predictor specifically for switch-type statements. A table of previous branch targets is maintained, indexed by the operand to the switch statement (in our case the opcode for a VM instruction). This will give almost perfect indirect branch prediction for a switch-based interpreter. The case block table is designed specifically for switch statements, so it is less general than a two-level predictor, but it provides very high accuracy. To our knowledge, the case block table has not found its way into commercially available general-purpose processors.

Papers dealing with superoperators and superinstructions [Proebsting 1995; Piumarta and Riccardi 1998; Hoogerbrugge et al. 1999; Ertl et al. 2002] 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 the work of Ertl *et al.* [2002]). 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, usually resulting in a slowdown.

The problem of creating dynamic superinstructions out of *basic blocks* which contain quickable instructions has been tackled previously using *preparation sequences* [Gagnon and Hendren 2003; Gagnon 2003]. In this approach, basic blocks which do not contain any quickable instructions are used to construct dynamic superinstructions as per usual. For basic blocks that contain quickable instructions, a bytecode routine (the preparation sequence) is added to the end of the method containing the relevant block. The actual basic block in the bytecode is replaced with a stub to call this preparation sequence, which performs the work of the basic block and additionally, when all instructions in the basic block have been quickened, replaces the stub in the bytecode with the address of this dynamic superinstruction.

This contrasts with our approach, where we leave an appropriately sized gap in the dynamic superinstruction for the code for the quick version of the instruction. This gap is partially filled by a copy of the VM instruction dispatch code, which dispatches to the quickable version of the code. As part of the quickening process, code for the quick version of the instruction is patched into this gap, replacing the dispatch. Locks are used to avoid race conditions in the quickening process.

Preparation sequences are attractive, given that they provide a lock-free, thread safe technique for dealing with quickable instructions when creating dynamic superinstructions. However, preparation sequences solve a slightly different problem, namely constructing dynamic superinstructions composed from single basic blocks. Creating dynamic superinstructions that extend across basic block boundaries is a more difficult problem. In particular, a basic block has only a single entry point, so the threaded code can be updated to use the superinstruction with a single, atomic store. A superinstruction extending across basic blocks must have multiple entry points, because different parts of the superinstruction are targets for taken VM branches (which are implemented with real machine indirect branches), so multiple points in the threaded code must be updated. Thus, we must use a more complex locking scheme, rather than the elegant lock-free preparation sequences.

A similar approach to preparation sequences was recently implemented in the interpreter in the Cacoa Java VM [Ertl et al. 2006].

9. CONTRIBUTIONS

In this paper, we have looked at software ways to improve prediction accuracy of interpreters on BTBs. The main contributions of this paper are:

—We propose a new technique, termed *replication* (Section 4.1), for eliminating mispredictions.

—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 describe implementations of these techniques in interpreters for both Java and Forth. There are a number of complicating factors in implementing these techniques for Java, and we describe our solutions to these problems (Section 5.4).

—We empirically compare the static [Ertl et al. 2002] and dynamic [Piumarta and Riccardi 1998] superinstruction techniques against each other (Section 7).

The use of these techniques can result in significant speedups for VM interpreters by reducing the number of indirect branch mispredictions that occur during execution.

10. 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 two 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 3.07 over an efficient threaded-code interpreter); and combining sequences of VM instructions into superinstructions (speedup up to a factor of 3.39). In combination these techniques achieve an even greater speedup (up to a factor of 4.55).

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 2.35), 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 4.30), 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 4.55).

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.

ACKNOWLEDGMENTS

We thank the PLDI 2003 referees for their helpful comments on an earlier version of this work, and the anonymous reviewers of TOPLAS who helped us to greatly improve this paper. The performance counter measurements were made using Mikael Pettersson's perfctr package and the papiex tool written by Philip J. Mucci.

REFERENCES

Bell, J. R. 1973. Threaded code. *Commun. ACM 16,* 6, 370–372.

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

Berndl, M., Vitale, B., Zaleski, M., and Brown, A. D. 2005. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *3nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2005), 20-23 March 2005, San Jose, CA, USA*. 15–26.

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

Casey, K., Ertl, A., and Gregg, D. 2005. Optimizations for a Java interpreter using instruction set enhancement. Tech. Rep. TCD-CS-2005-61, Department of Computer Science, University of Dublin, Trinity College, Dublin 2, Ireland. September.

Casey, K., Gregg, D., and Ertl, A. 2005. Tiger - an interpreter generation tool. In *International Conference on Compiler Construction (CC 05)*. LNCS 3443. Springer Verlag, Edinburgh, Scotland, 246–249.

Casey, K., Gregg, D., Ertl, M. A., and Nisbet, A. 2003. Towards superinstructions for Java interpeters. In *Proceedings of the 7th International Workshoop on Software and Compilers for Embedded Systems (SCOPES 03)*, A. Krall, Ed. LNCS 2826. Vienna, Austria, 329–343.

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

Driesen, K. and Hölzle, U. 1999. Multi-stage cascaded prediction. In *EuroPar'99 Conference Proceedings*. LNCS, vol. 1685. Springer, 1312–1321.

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

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

Ertl, M. A. and Gregg, D. 2003b. The structure and performance of *Efficient* interpreters. *The Journal of Instruction-Level Parallelism 5*. http://www.jilp.org/vol5/.

Ertl, M. A. and Gregg, D. 2006. Optimizing Interpreters for Processors with Branch Target Buffers. Tech. Rep. TCD-CS-2006-51, Department of Computer Science, University of Dublin, Trinity College, Dublin 2, Ireland. September.

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

Ertl, M. A., Thalinger, C., and Krall, A. 2006. Superinstructions and replication in the Cacao JVM interpreter. *Journal of .NET Technologies 4,* 1, 31–38.

Gagnon, E. 2003. A portable research framework for the execution of Java bytecode. Ph.D. thesis, McGill University.

Gagnon, E. and Hendren, L. J. 2003. Effective inline-threaded interpretation of Java bytecode using preparation sequences. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*. 170–184, volume 2622.

Gagnon, E. M. and Hendren, L. J. 2001. SableVM: A research framework for the efficient execution of Java bytecode. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01)*. Monterey, California, USA, 27–39.

Gochman, S., Ronen, R., Anati, I., Berkovits, A., Kurts, T., Naveh, A., Saeed, A., Sperber, Z., and Valentine, R. 2003. The Intel Pentium M processor: microarchitecture and performance. *Intel Technology Journal 7,* 2 (May), 20–36.

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

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

Kaeli, D. R. and Emma, P. G. 1994. Case block table for holding multi-way branches. US Patent No. 5,333,283.

Kaeli, D. R. and Emma, P. G. 1997. Improving the accuracy of history-based branch prediction. *IEEE Transactions on Computers 46,* 4, 469–472.

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

Krall, A. 1994. Improving semi-static branch prediction by code replication. In *Conference on Programming Language Design and Implementation*. SIGPLAN, vol. 29(7). ACM, Orlando, 97–106.

Li, T., Bhargava, R., and John, L. K. 2005. Adapting branch-target buffer to improve the target predictability of Java code. *ACM Transactions on Architecture and Code Optimization 2,* 2 (June), 109–130.

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

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

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

Rossi, M. and Sivalingam, K. 1996. A survey of instruction dispatch techniques for byte-code interpreters. Tech. Rep. TKO-C79, Faculty of Information Technology, Helsinki University of Technology. May.

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

Smith, J. and Nair, R. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann.

Sun-Microsystems. 2001. The Java Hotspot virtual machine. Tech. rep., Sun Microsystems Inc.

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

Young, C. and Smith, M. D. 1994. Improving the accuracy of static branch prediction using branch correlation. In *Achitectural Support for Programming Languags and Operating Systems (ASPLOS VI)*.

Zhou, J. and Ross, K. A. 2004. Buffering database operations for enhanced instruction cache performance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM Press, Paris, 191–202.

Received October 2005; revised October 2006; accepted March 2007.