The Behavior of *Efficient* Virtual Machine Interpreters on Modern Architectures

M. Anton Ertl
Institut für Computersprachen,
TU Wien, A-1040 Wien,
anton@complang.tuwien.ac.at

David Gregg
Department of Computer Science,
Trinity College, Dublin 2, Ireland.
David.Gregg@cs.tcd.ie

Abstract. Romer et al (ASPLOS 96) examined several interpreters and concluded that they behave much like general purpose integer programs such as gcc. We show that there is an important class of interpreters which behave very differently. Efficient virtual machine interpreters perform a large number of indirect branches (3.2%–13% of all executed instructions in our benchmarks, taking up to 61%-79% of the cycles on a machine with no branch prediction). We evaluate how various branch prediction schemes and methods to reduce the mispredict penalty affect the performance of several virtual machine interpreters. Our results show that for current branch predictors, threaded code interpreters cause fewer mispredictions, and are almost twice as fast as switch based interpreters on modern superscalar architectures.

1 Introduction

In recent years, interpretive language implementations have become commonplace for many tasks. Despite the importance of interpreters, little effort has been devoted to measuring their behavior on modern pipelined and superscalar architectures. The main work on this topic is by Romer et al. [9] who studied several interpreters (MIPSI, Java, Perl, Tcl). From their measurements, they drew two important conclusions. First, that the interpreters they measured behaved very much like general purpose integer C programs such as gcc, and were unlikely to benefit from any hardware support. Secondly, the interpreters they examined were not particularly efficient. From their experiments, and from examining the code of the interpreters, it was clear that performance was not a central goal of the interpreters' writers. Romer et al concluded that interpreters are likely to benefit far more from an efficient implementation than from hardware support.

In this paper we look at an important class of interpreters which behave very differently. These are efficient virtual machine interpreters. We find that they perform an exceptionally high number of indirect branches. Typical C code performs significantly less than 1% non-return indirect branches; C++ programs (using virtual function calls) perform 0.5%–2% indirect branches [4]; and other interpreters perform less than 1.5% non-return indirect branches. But up to 13% of the instructions executed in the virtual machine interpreters we examine are non-return indirect branches. Consequently, the performance of efficient virtual machine interpreters is highly dependent on the indirect branch prediction accuracy, and the branch misprediction penalty. For one benchmark we see a speedup factor of 2.55 or more (depending on the mispredict penalty) between no predictor and a good predictor.

The main contributions of this paper are (1) to demonstrate that (unlike many interpreters) efficient virtual machine interpreters do not behave like general purpose C programs, (2) to quantify the effects that indirect branches have on such interpreters on modern hardware, and (3) to show that different implementation techniques, especially using a threaded code interpreter, can greatly reduce the effects of indirect branches.

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

2 Efficient virtual machine interpreters

For an efficient interpreter for a general purpose language the design of choice is a virtual machine (VM) interpreter. The program is represented in an intermediate code that is similar in many respects to real machine code: the code consists of VM instructions that are laid down sequentially in memory and are easy to decode and process by software. Efficiently implemented virtual machine interpreters perform well even on programs consisting of many simple operations (5–10 times slower than optimized C, with another factor of 2–3 for run-time type-checking).

The interpretation of a virtual machine instruction consists of accessing arguments of the instruction, performing the function of the instruction, and dispatching (fetching, decoding and starting) the next instruction. The most efficient method for dispatching the next VM instruction is direct threading [1]. Instructions are represented by the addresses of the routine that implements them, and instruction dispatch consists of fetching that address and branching to the routine. Direct threading cannot be implemented in ANSI C and other languages that do not have first-class labels, but GNU C provides the necessary features. Implementors who restrict themselves to ANSI C usually use the giant switch approach (2): VM instructions are represented by arbitrary integer tokens, and the switch uses the token to select the right routine.

When translated to machine language, direct threading typically needs three to four machine instructions to dispatch each VM instruction, whereas the switch method needs nine to ten [6]. The additional instructions of the switch method over direct threading is caused by a range check, by a table lookup, and by the branch to the dispatch routine generated by most compilers.

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

  goto *ip++;

 add:
  sp[1]=sp[0]+sp[1];  sp++;  goto *ip++;
}
```

Fig. 1. Direct threading using GNU C's "labels as values"

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

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

Fig. 2. Instruction dispatch using `switch`

3 Benchmarks

This section describes the interpreters we have benchmarked. It also highlights the differences between the efficient VM intrepreters and the Perl and Xlisp interpreters, which behave more like general purpose C programs. (These benchmarks can be found at `http://www.complang.tuwien.ac.at/anton/interpreter-arch`).

Gforth 0.4.9-19990617, a Forth system [5]; it uses a virtual stack machine. We compiled it for indirect threaded code [3] i.e., there is one additional load instruction between the instruction load and the indirect branch (the direct threaded version would not compile for Simplescalar). The workloads we use are *prims2x*, a compiler for a little language, and *bench-gc*, a conservative garbage collector.

Ocaml 2.04, the Objective CAML "bytecode" interpreter, which also implements a virtual stack machine [7]; the VM works on tagged data (with some associated overhead), but does not do run-time type-checking (the tags are probably for the garbage collector). We ran both, a version using direct-threaded code, and a switch-based version. The workloads we use are *ocamllex*, a scanner generator, and *ocamlc*, the Ocaml compiler.

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

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

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

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

We use Gforth, Ocaml, Scheme48, and Yap as examples of efficient VM interpreters. We had hoped to include a Java VM interpreter, but we couldn't find one that was both really efficient and freely available. Apparently everyone is focusing on JIT compilers. We include Perl and Xlisp for comparison with existing work [4], and to demonstrate the difference between many commonly used interpreters and efficient VM interpreters. We selected workloads of non-trivial size and used two workloads for most interpreters in order to exercise significant portions of the interpreters and to avoid overemphasizing some non-representative aspect of the performance. All benchmarks (except Scheme48) were run to completion.

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

Fig. 3. Instruction class distribution for our benchmarks

We compiled the interpreters for the SimpleScalar 2.0 microprocessor simulator [2], which simulates a MIPS-like architecture. Some basic data on the behavior of these benchmarks is listed in Fig. 3.

The aspect of these data that is important for this paper is the extremely high proportion of indirect branches in the efficient interpreters: 3.2%–13%. Note also that Gforth requires one load less per indirect branch if it uses direct-threaded instead of indirect-threaded code, resulting in an even higher proportion of indirect branches. In comparison, the benchmarks used in indirect branch prediction research [4] perform at most 2.1% indirect branches.

The reason for all these indirect branches is that every virtual machine instruction performs at least one indirect branch, whether it uses threaded code or switch dispatch. Most virtual machine instructions are quite simple (e.g., add the top two numbers on the stack), and can be implemented in a few native instructions plus dispatch code.

Another issue that can be seen nicely is the effect of threaded code vs. switch dispatch on the instruction count. The threaded and switch based variants of the interpreters were created by using existing compile time options in the interpreters, which use conditional compilation to create the appropriate version. The number of indirect branches is exactly the same in both cases, but the total number of instructions is higher for interpreters using switch dispatch (by a factor of 1.76 for ocamllex). For Ocaml switch dispatch costs 5.9–6.6 instructions more than threaded code, for Yap the difference is 7.9 instructions.

4 Fast indirect branches

Unless special measures are taken branches are relatively expensive operations in modern, pipelined processors. The reason is that they typically only take effect after they have reached the execute stage (stage 10 in the P6 micro-architecture, stage 17 in the Pentium 4) of the pipeline, but their result affects the instruction fetch stage (stage 1); i.e., after such a branch the newly fetched instructions have to proceed through the pipeline for many cycles before reaching the execute stage. The resulting delay is known as the misprediction penalty.

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

Profile-Guided Prediction The Alpha architecture supports profile-guided, static indirect branch prediction by having a field in the JMP instruction that specifies 16 bits of the predicted branch target (this should be enough to predict an instruction in the I-cache). However, exploiting this feature requires compiler support for profile feed-back. Unfortunately, the compilers we know don't support this in a way that helps interpreters.

Dynamic Prediction Dynamic branch prediction is a micro-architectural mechanism and works without architectural, compiler or software support. The simplest dynamic indirect branch prediction mechanism is the branch target buffer

(BTB), which caches the most recent target address of a branch; it uses the address of the branch instruction as the key for the lookup.

An improvement on the normal BTB is the BTB with 2-bit counters, which replaces a BTB entry only when it mispredicts twice. Actually one bit is sufficient, if the BTB is used only for indirect branch prediction. This halves the mispredictions if a branch usually jumps to just one target, with just a few exceptions.

The best currently-known indirect branch prediction mechanisms are two-level predictors that combine a global history pattern of n indirect branch targets with the branch address and use that for looking up the target address in a table; these predictors were proposed by Driesen and Hölzle [4], and they studied many variations of these predictors. We discuss the effects of various prediction mechanisms on interpreters in Section 5.

5 Evaluation

We added several branch prediction methods to the SimpleScalar-2.0 simulator, and ran the interpreters with the following prediction methods. In all cases, we used an unlimited sized table for the branch predictor, since we are interested in measuring the predicability of the branches with a given model, rather than tuning existing predictors.

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

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

BTB Predicts that a branch always jumps to the target it jumped to last time.

BTB2 A BTB that only replaces the prediction on the second miss.

2-Level Two-Level predictors that use the branch address and a history register containing the last 1–8 target addresses as a key into a lookup table of predicted targets and two-bit-counters. We use all the bits of the address. The history register is updated speculatively with the predicted target when the instruction fetch from the predicted target occurs.

Figure 4 shows the mispredict rates of our efficient VM interpreters; Fig. 5 shows the resulting execution time in cycles per VM instruction, for SimpleScalar with the shortest mispredict penalty; Fig. 6 shows the execution time when the misprediction penalty is set to 13 cycles (i.e., a Pentium-III-like pipeline length). Note that no results appear for Xlisp or Perl in these two figures, because they are not VM interpreters. Finally, Fig. 7 shows the number of machine cycles per non-return indirect branch executed.

Even with the smallest mispredict penalty the difference between using the best and the worst predictor can result in a speed difference of 1.36 (yap-boyer) up to 2.55 (gforth-benchgc); with the 13 cycle misprediction penalty the speed differences rise to 1.74–4.77 (yap-chat-sw–gforth-benchgc). So, indirect branch

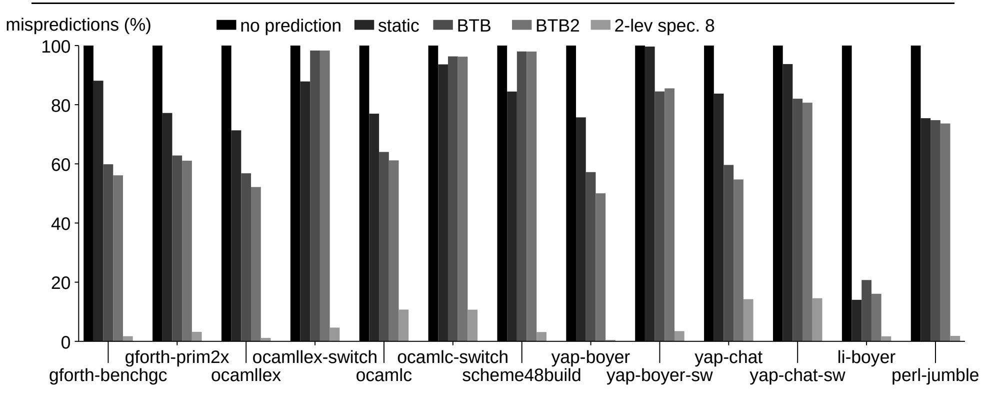

Fig. 4. Mispredict rate of various predictors on interpreters

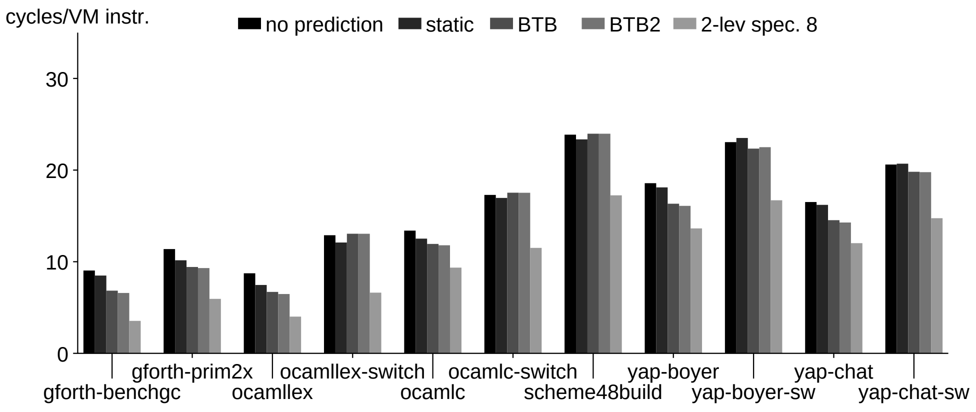

Fig. 5. Execution time of various interpreters with various branch prediction scheme with a mispredict penalty of 4 cycles ("1 cycle" in SimpleScalar terminology)

prediction accuracy plays a major role in VM interpreter performance (even more than the instruction counts indicate).

For threaded code, we see 71%–88% mispredictions for profile-guided static prediction, 57%–63% mispredictions for BTB, and 50%–61% for BTB2. With a low mispredict rate, gforth-benchgc takes 3.5 cycles per VM instruction and ocamllex takes 4 cyles, showing that threaded-code interpreters can be very fast, if we can eliminate most mispredictions.

For switch dispatch, we see 84%–100% mispredictions for profile-guided static prediction. The reason for the difference to the threaded-code case is that the switch-based interpreter uses a shared dispatch routine common to all instructions; profile-guided static prediction can therefore predict only the most common VM instruction for the whole training program; with a dispatch routine per VM instruction, profile-guided prediction can predict different next VM instruc-

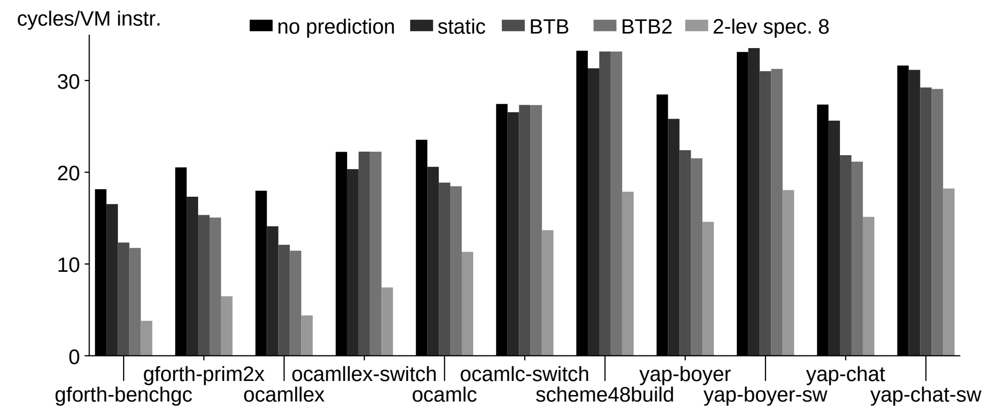

Fig. 6. Execution time of various interpreters with various branch prediction scheme with a mispredict penalty of 13 cycles ("10 cycles" in SimpleScalar terminology)

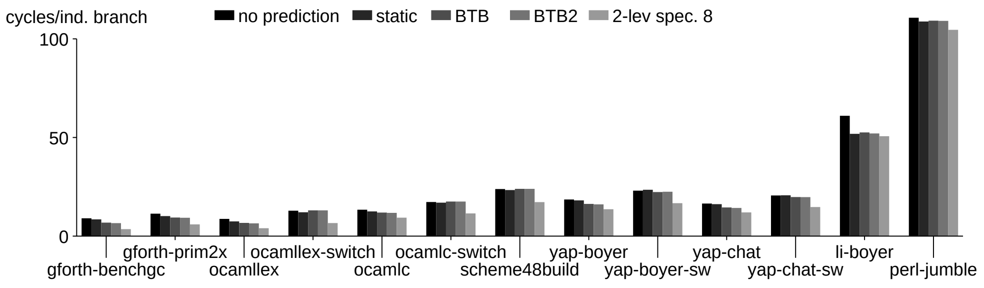

Fig. 7. Execution cycles per non-return indirect branch, with a misprediction penalty of 4 cycles ("1 cycle" in SimpleScalar terminology)

tions, one for each current VM instruction. We expect mispredict rates similar to the threaded-code case, if the C compiler replicates the switch code for every VM instruction. The 100% misprediction result (yap-boyer-sw) is apparently due to differences in VM instruction usage between the training and the reference program.

For BTB and BTB2 we see 98% mispredictions for ocaml and scheme48, and 81%–86% for Yap. Again, the reason for the differences from the threaded-code results is the shared dispatch code: With separate jumps the BTB works quite well if each VM instruction occurs at most once in a loop of the interpreted program; then the BTB will predict all the indirect jumps correctly on the second trip through the loop (and BTB2 on the third trip). With a shared dispatch routine, a BTB will just predict that the present instruction will occur again (which seems to happen about 15% of the time in Yap's register-based VM, but hardly ever in the stack-based VMs of Ocaml and Scheme48).

For the two-level predictors the difference between switch dispatch and threaded code is smaller, probably because the two-level predictors can fall back on the history and are not limited to looking at the address of the current branch.

The overall effect of these differences is that on SimpleScalar with 4 cycles mispredict penalty, switch dispatch takes 4.1–4.5 cycles more without prediction than threaded code, 5.5–6.6 cycles more for BTB2, and 2.6–3.1 cycles more for 2-level spec.(8). Given the short execution time of the threaded-code VM instructions, using switch dispatch is slower by up to a factor of 2.02 (ocamllex with BTB2 and 4 cycles mispredict penalty). Thus threaded code not only needs fewer machine instructions to execute, it also causes substantially fewer mispredictions.

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

6 Improving Interpreters

To increase the effectiveness of BTBs, each indirect branch should, ideally, have exactly one target, at least within the working set (e.g., an inner loop). The problem with the switch-based interpreters is that there is only a single branch, and usually there are more than one different VM instructions (and thus branch targets) in the working set. Threaded code interpreters improve the accuracy by having one branch per VM instruction; if a VM instruction occurs only once in the working set, its branch will correctly predict its target (the code for the next instruction) all the time.

Another approach is to increase the number of instructions in the VM instruction set. Several optimisations already do this. For example, splitting frequent VM instructions into several variants, preferably specialising them for frequent immediate arguments. Combining common sequences of VM instructions into single “super” instructions [8] will also add context information, and has the added advantage of removing dispatch overhead (and branches) within the sequence. Such specialised instructions should have a smaller number of frequent successors in most programs, and so their indirect branch will be more accurate.

7 Conclusion

Efficient virtual machine interpreters execute a large number of indirect branches (up to 13% of instructions in our benchmarks). Without indirect branch prediction, the resulting mispredictions can take most of the time (up to 62%) even on a processor with a short pipeline (and almost 80% with a longer pipeline). Even with indirect branch prediction, mispredicion rates are remarkably high. Profile guided static prediction only yields an average accuracy of 11%. Branch target buffers give accuracies of 2% to 50%, with a slight improvement for the two-bit variant. Two level predictors increase the performance of efficient VM interpreters significantly, by increasing prediction accuracy to 82%–98%.

Threaded code interpreters are much more predictable than switch based ones, increasing accuracy from 2%–20% to about 45% on a BTB2. The reason is that a switched based interpreter has only a single indirect branch jumping to many targets, whereas a threaded code interpreter has many branches, each of them jumping to a much smaller number of frequent targets. Given that threaded code interpreters also require less overhead for instruction dispatch, they are clearly the better choice for efficiently implementing VM interpreters on modern processors (up to a factor of two speedup for our benchmarks).

Acknowledgements

Vitor Santos Costa was a great help in getting Yap to run on SimpleScalar.

References

1. James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.
2. Douglas C. Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.
3. Robert B.K. Dewar. Indirect threaded code. *Communications of the ACM*, 18(6):330–331, June 1975.
4. K. Driesen and U. Hölzle. Accurate indirect branch prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, volume 26,3 of *ACM Computer Architecture News*, pages 167–178, New York, June 27–July 1 1998. ACM Press.
5. M. Anton Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, Mariánské Láznè (Marienbad), 1993.
6. M. Anton Ertl. Stack caching for interpreters. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315–327, 1995.
7. Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.
8. Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Principles of Programming Languages (POPL '95)*, pages 322–332, 1995.
9. Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The structure and performance of interpreters. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 150–159, 1996.