Primitive Sequences in General Purpose Forth Programs

David Gregg * John Waldron
Department of Computer Science
Trinity College Dublin
David.Gregg@cs.tcd.ie

Abstract

Instruction dispatch is responsible for most of the running time of Forth interpreters, especially on modern pipelined processors. Superinstructions are an important optimisation to reduce the number of instruction dispatches. Superinstructions have been used for many years to optimise interpreters, but an open problem is the choice of superinstructions to include in the interpreter. In this paper we propose a number of heuristics for choosing superinstructions, and evaluate them for general purpose Forth programs. We find that static measures of frequency perform well for superinstruction selection. As few as eight superinstructions can reduce the number of instruction dispatches by an average of 15%, and reductions of up to 45% are possible with large numbers of superinstructions.

1 Motivation

Traditionally, Forth is implemented with an interpreter. Interpreters have many advantages over compilers. They are simpler than compilers, which can make them more reliable and easier to maintain. An interpreter written in a high-level language can be made easily portable to new architectures. In addition, interpreters along with their code can require less memory than compiled machine code. For these reasons, many Forth implementations, such as GForth [Ert93] are based on interpreters rather than compilers.

Although interpreters are superior to compilers in most respects, their major drawback is that interpreted code runs much more slowly than compiled code. This problem is particularly acute on modern pipelined processors, which rely on branch prediction to keep their pipelines full.

The code for a virtual machine (VM) interpreter is laid out in memory just like real machine code. To execute a virtual machine instruction (that is, a Forth primitive) the interpreter must branch to one of a large number of different pieces of code, depending on what type of instruction is to be executed. This is known as dispatching the instruction. The multi-way branch is implemented in the interpreter using a `switch` statement, threaded dispatch (which requires a language which allows labels as first class values, such as GNU C), or function pointers [Ert96]. Regardless of which method is used in the source code, the multi-way branch becomes an indirect branch when the interpreter is compiled into machine code. Current branch predictors deal poorly with indirect branches. For example, Ertl and Gregg [EG01] found that current predictors mispredict 60%–97% of indirect branches in interpreters, and that many interpreters spend up to 70% of their time in branch mispredictions.

A number of solutions to this problem have been proposed. One is to wait for processors with better indirect branch predictors. Two-level indirect branch predictors [DH98a, DH98b, KK98] can increase the prediction accuracy to more than 98% on interpreters [EG01]. However, most programs execute only a tiny proportion of indirect branches (typically less than 0.5% of executed instructions, and very rarely more than 2% [DH98a]) so processors with better indirect branch predictors are unlikely to appear soon. The benefit is simply too small for almost all programs other than interpreters.

Another possible solution is to *software pipeline* the interpreter [HATvdW99] by moving part of the dispatch code (which may include long latency loads) for future VM instructions into the current instruction. If the dispatch indirect branch is mispredicted, any long latency instructions that started before the branch can run to completion during the time of the branch misprediction penalty. Ertl et al [EGKP02] evaluated the benefit of this type of prefetching, and found that it decreased running time of the Gforth interpreter by an average of about 10%. The main problem with this technique is that branch misprediction penalties are typically very long (10-20 cycles), whereas the latency even of loads is typically no more than a few cycles, so there is not much useful work than can be over-

*Supported by Enterprise Ireland International Collaboration Programme, Project IC/2002/167

lapped with a branch misprediction.

A third solution is to use superinstructions. A superinstruction is a new VM primitive instruction which behaves exactly like a sequence of regular VM primitives. The superinstruction is implemented as a single VM instruction, however, so the interpreter overhead of executing it is very much lower than for the original sequence.

There are two main types of superinstructions. *Dynamic superinstructions* [Piu98] are new primitives which are generated when the interpreter is already running. The set of dynamic superinstructions can be tailored to the specific running program, so choosing a good set of superinstructions is easy. However, the interpreter must be constructed using a language or system that supports run-time code generation or copying. At the very least, there must be some way to create new executable code in main memory, to synchronise the instruction cache with the code in main memory (which typically requires some machine specific assembly language code), and to jump to the new code from existing code. These features all reduce the portability and simplicity of the interpreter, and are not trivial to implement.

Static superinstructions are superinstructions that are chosen and added to the interpreter at the time of its construction. Static superinstructions can be specified in exactly the same way as any other virtual machine primitive. They do not require that the interpreter be written in a language with any special features, or machine specific code. However, the set of static superinstructions must be chosen when the interpreter is constructed, most likely at a time when one doesn't know which programs will be run on the interpreter. Ertl et al. [EGKP02] found that static superinstructions can increase the speed of a Forth interpreter by almost a factor of two. The remainder of this paper deals with static superinstructions.

Static superinstructions have been used for many years to optimise interpreters, but there are still a number of unsolved problems with their use. One such problem is the choice of superinstructions to include in the interpreter. The set of superinstructions must be chosen when the interpreter is constructed, at a time when the workload is probably not known. Clearly, it is desirable to choose sequences that are likely to appear in many programs, preferably in the inner loops. One also wants to avoid cluttering the interpreter with unused superinstructions.

In this paper, we examine strategies for choosing superinstructions for general purpose Forth programs. In compiler and computer architecture research it is common to divide programs into several different types [HP90]. Computer architectures and compiler optimisations are typically designed for a particular type of code. For example, scientific code, such as whether forecasting and nuclear weapon simulation, typically consists of large loops containing array and floating point computations. Scientific code usually contains few `if` statements. DSP code, such as music or image filters, is similar, but loops tend to iterate a smaller number of times. Database code tends to contain large numbers of `if` statements, the outcome of which is dependent on user input and the information in the database. Another important feature of database code is that it has very poor locality of reference; little time is spent in tight loops.

Perhaps the most important type of widely used code is general purpose integer code. This type of code includes operating systems, word processors, compilers, database programs and many of the other types of programs that typically run on a desktop machine. This code is similar to database code in that it tends to contain large numbers of `if` statements, and control flow depends heavily on user input. General purpose integer code spends more times in loops that database code however. Loops iterate a (widely varying) average of about ten times, and the number of iterations is usually dependent on user input.

In this paper we consider the problem of choosing a set of useful superinstructions for general purpose integer programs written in Forth. It is important to note that at the time we choose the superinstructions we do not know which programs will run on the Forth system. When sample programs are used as the basis for choosing a set of superinstructions, we always use a different set of programs for selecting superinstructions from the programs used to evaluate the usefulness of the selection. We consider several criteria for choosing sequences of instructions from sample programs, such as their static frequency in the code, and the number of times that they are executed dynamically. All strategies are tested experimentally, and results are presented for the most effective ones.

The remainder of this paper is organised as follows. We first describe existing work on superinstructions for interpreters (section 2). Section 3 introduces our strategies for choosing superinstructions. In section 4 we describe five general purpose programs we use for our experiments. Section 5 presents our system for evaluating various superinstruction selection strategies. In section 6 we present an empirical evaluation of the different design choices. Finally, in section 7 we draw conclusions.

2 Related Work

Programmers of interpreters have combined instructions into superinstructions for many years. However, the earliest published reference on such a combining system that we are aware of is from Schütz [Sch92].

In the past superinstructions made interpreters more difficult (expensive) to maintain. The reason is that superinstructions were added manually to the interpreter. Interpreter generators [Pro95] allow superinstructions to be generated automatically. Typically, profiling data from a set of test programs is used to choose a list of superinstructions. This list is fed into the interpreter generator, which automatically combines the code to implement each of the VM instructions. More sophisticated interpreter generators, such as vmgen [EGKP02], can also perform stack optimisations.

An alternative to combining sequences of instructions is to combine operators earlier in the process of compiling a program. Hughes [Hug82] proposes such a scheme, which operates on the parse tree for a program. It identifies common combinations of operators, and combines them into "super-combinators". If represented as a sequence of instructions, a super-combinator may have one or more gaps which can contain other VM instructions. A similar scheme is proposed by Proebsting [Pro95].

In previous, unpublished work Ertl measured the frequency of instructions, and sequences of instructions in three large Forth programs. The raw data from these measurements can be found at `http://www.complang.tuwien.ac.at/forth/peep/`.

The best developed body of related work is on dictionary-based text compression [BCW90]. Dictionary compression schemes reduce the size of a text by replacing common sequences in the text with references to phrases in a dictionary. Superinstructions are an application of dictionary-based compression, where the dictionary is fixed in advance, compression must stop at basic block boundaries, and the emphasis is on compressing the most commonly executed basic blocks.

Research on compression theory has found a number of results which are directly relevant to choosing superinstructions. Perhaps the most important of these is that for a given text, choosing an optimal set of phrases for a dictionary to compress that text is NP-hard. Thus, choosing the best set of superinstructions for a given sample of programs is also NP-hard. A practical scheme for choosing superinstructions must be based on heuristics. In the next section, we present a number of promising heuristics.

One branch of text compression which is similar to our work is on split stream dictionary compression [Luc00], which is used for compressing bytecode programs. A bytecode program consists of a mixture of opcodes and operands. Typically, there is a strong pattern in sequences of opcodes. For example, in Java bytecode, an opcode to load a pointer to an object is often followed by a bytecode to invoke a method on that object. However, the opcodes are interspersed with operands to specify from which local variable the pointer should be loaded, and which method should be invoked. These operands may also follow a pattern. By splitting the bytecode into two streams — one for each of opcodes and operands, better compression can be achieved than by mixing them together.

An alternative to improving interpreters is to compile directly to native code. Bruno and Lassagne [BL75] describe a system for generating optimal code for stack machines. Rose [Ros86] presents a "subroutine threaded" implementation of Forth which expresses programs as a sequence of calls to subroutines which implement the primitives. Almy [Alm87] describes further optimisations for Forth.

3 Superinstruction Selection

The most obvious way to select superinstructions is to choose the most common sequences appearing in a set of sample programs. As we show in section 6 this is an effective strategy. However, there are a number of ways that the most common sequences could be defined. Furthermore, choosing one sequence might affect the desirability of another. In this section, we explore a number of criteria that can be used in choosing superinstructions.

The most basic decision in a superinstruction selection strategy is whether to choose sequences based on how often they appear statically in sample programs, or the frequency with which they are executed dynamically. Intuitively, it seems better to choose dynamically frequent sequences. If a sequence is executed frequently in one program, it is perhaps likely to be executed frequently in another.

The counter-argument is that programs spend most of their time in a handfull of loops. The sequences in these loops are so much more frequently executed that the other code in the program that sequences from other code will be selected only after every sequence within these loops has been chosen. Thus, one is trying to choose common sequences based primarily on a handfull of small samples. These loops are often peculiar to the particular program, and result in sequences that are close to optimal for the sample program, but useless for other purposes. Statically frequent sequences, on the other hand, are based on a much larger sample of code, and should represent genuinely frequent code patterns. The weakness of static measures is that some Forth primitives, such as `(loop)`, are dynamically frequent, but statically extremely rare [GEW01]. A compromise solution is to use some sort of hybrid.

A second important decision is whether or not to weight the sequences by length. A longer sequence reduces the number of dispatches by a larger amount, and thus gives a greater benefit. On the other hand, even if a longer sequence occurs frequently in the training programs, it is less likely to be used in practice. The longer the sequence, the more likely that it is peculiar to a particular training program.

A third important decision is whether choosing one sequence to be a superinstruction should affect the choice of other sequences. The most straightforward strategy is to simply order all sequences by frequency, and choose those with the highest rank. However, the value of a sequence such as `lit @ lit @` is lower if one has already chosen the sequence `lit @ lit` as a superinstruction. One might be better to choose a completely different sequence.

Another consideration arises when one combines data from several different programs. Using several different programs to choose frequent sequences is highly desirable, since it decreases the likelihood of choosing sequences that are only useful for the training program. A problem is that one program may run for much longer than another. If the data for the two programs are simply combined, the long running program will dominate the dynamic results. A similar problem arises when one program is statically much larger than another. One solution to this problem is to normalise the results for each program. For example, the frequency of each sequence could be expressed as a percentage of total instructions in the program. When combining data from different programs, the percentages could be combined rather than the absolute numbers. In the following sections we evaluate these strategies, and present results using five general purpose programs.

4 Benchmarks

This section describes the benchmarks that we use to evaluate the strategies for choosing superinstructions. We have chosen five large, general purpose programs for our experiments, which we believe are generally representative of real programs.

We deliberately chose not to use small, artificial benchmark programs for our evaluation. The main reason is that such programs behave very differently from larger ones [GEW01], so any conclusions that we drew about them might not be applicable to real programs. In addition, choosing superinstructions that are useful for micro-benchmarks is not easy, unless one knows at the time of constructing the interpreter that it will be used to run that particular microbenchmark. Typically, these programs spend virtually all their time in a single small loop. The usefulness of superinstructions for these programs depends almost entirely on whether one happens to choose one or more sequences that match the handfull of instructions in the loop.

It is easier to select useful superinstructions for a larger program, since the execution time is typically spread over several loops. A representative set of superinstructions is more likely to find matches in a larger body of code. Thus, we restrict ourselves to the easier problem of choosing superinstructions for real general purpose programs[1]. Basic information about the benchmarks appears in Fig. 1. The benchmarks are:

prims2x A virtual machine interpreter generator which forms part of the Gforth system. It accepts a specification of the virtual machine instructions and outputs C source for an interpreter.

gray A parser generator which accepts an LL1 grammar and produces a recursive descent parser in Forth.

brew An evolutionary programming playground, which simulates the interaction between creatures.

brainless A chess playing program.

benchgc A conservative garbage collector for Forth. It was run with a test program which allocates and collects large amounts of memory.

5 Experimental Setup

The benchmarks were measured with the Gforth system [Ert93]. Gforth is a complete, product quality implementation of the ANS Forth standard which is freely available under the GNU general public licence. Gforth is an interpreter based implementation of Forth, which allows the Forth engine to be simple and portable.

We modified the Forth text interpreter (which compiles Forth source to threaded virtual machine code) and the engine interpreter (which interprets the threaded code) to collect information about running programs. The most important modification

[1] Note that in previous work [GEW01], we also examined the behaviour of another large program, pentomino. However, pentomino generates very large amounts of almost identical code at run time, and could not, by any normal measure, be described as a general purpose program.

Benchmark	source lines	static primitives	dynamic primitives.
prims2x	1258	6,314	18,319,272
gray	1458	4,792	4,833,582
brew	7627	6,078	1,312,698,495
brainless	3755	11,430	859,030,440
benchgc	1479	4,105	559,786,379

Figure 1: The benchmark programs

was to add additional code to all control flow instructions, so that they record the number of times that each basic block is entered.

One complication with measuring the behaviour of Forth code is that there is no "program" as such. A Forth system consists of a collection of words. Additional functionality is added by defining new words, in effect, changing the language. An important question when measuring Forth programs is whether to examine only the user-defined words, or whether to include both system and user-defined words. Should statistics for a small bubble sort benchmark include the entire Forth system or just the words in the user part of the code? The former would cause the system code to dominate static measurements of most programs, whereas the latter would leave out an important part of dynamic measurements. For this reason we chose a compromise, which is to include in our measurements those instructions from the system and user code which are executed at least once.

The simplest way to find common sequences of instructions up to a given length N is to modify the interpreter to keep a list of the most recent N instructions executed. After executing each instruction the list is added to a hash table which records the number of times that that sequence has occurred. A weakness of this approach is that the instructions in the resulting sequences may be from more than one basic block. Currently, sequence-based optimisations work only on instructions within a basic block. For this reason, we used a more sophisticated scheme to measure only those sequences which appear within basic block boundaries.

It is also important to note that some complex words appear in executable code that are expanded to primitives at run time. In Forth implementations other than GForth, these might be implemented by a single primitive. The replacements are shown in Fig. 2.

word	replacement
docol	call
docon	lit @
dovar	lit
douser	useraddr
dodefer	lit @ execute
dofield	lit +
dodoes	lit call

Figure 2: Replacements of words with sequences

When rewriting the program with superinstructions, there may be situations where there are several possible rewrites of a basic block. Identifying possible rewrites is known as *parsing* in compression terminology. Several different parsing strategies are possible, but for the experiments in this paper we used *longest match* parsing [BCW90]. This approach finds the longest sequence of primitives in the basic block that matches a superinstruction, and replaces them with that superinstruction. It then applies the same process again, until no more replacements are possible. Longest match parsing is simple to implement and gives results which are very close to optimal [BCW90].

In total we used five benchmark programs. When choosing superinstructions, we used four of these programs as training data. In other words, we chose the most frequent sequences from four programs, and then evaluated the usefulness of those superinstructions for the fifth program. We did this five times, each time using a different program to evaluate the superinstructions. Results are presented in the next section.

6 Results

To measure the effectiveness of each scheme, we initially computed the number of dispatches required to execute the program without superinstructions. We then chose the set of superinstructions using the given scheme, and rewrote the program using those superinstructions. We then recomputed the number of dispatches. The results show the percentage reduction in dispatches caused by replacing sequences of primitives with superinstructions. We implemented the following schemes.

static Sequences are chosen in order of static frequency.

dynamic Sequences are chosen in order of the

Scheme	8	16	32	64	128	256	512	1024	2048
static	14.81	17.51	21.11	24.45	28.54	33.62	37.22	41.71	44.24
static normalise	14.77	17.51	21.20	24.18	28.08	33.32	36.78	41.25	44.66
static length	13.47	16.77	20.74	24.06	26.77	30.28	36.56	38.53	42.40
static length normalise	13.47	16.77	20.34	23.82	26.25	29.16	33.99	38.71	42.18
static rewrite	14.83	17.91	19.99	23.40	28.98	34.14	37.78	42.58	44.27
static normalise rewrite	15.01	17.91	20.85	23.29	28.71	34.01	37.87	42.47	44.64
static length rewrite	14.61	18.22	20.08	22.83	27.02	31.89	36.36	41.32	42.38
static length normalise rewrite	14.32	17.87	19.50	22.23	25.38	31.93	35.67	41.25	42.95
dynamic	12.00	13.98	17.22	19.66	22.50	26.28	32.10	35.38	39.74
dynamic normalise	11.86	13.08	18.18	21.94	24.22	29.18	32.64	38.14	41.93
dynamic length	11.23	13.11	15.28	16.67	20.69	24.77	29.85	34.91	38.13
dynamic length normalise	10.86	13.72	16.23	18.96	22.89	26.61	30.94	35.86	41.19
dynamic rewrite	10.54	13.63	15.57	19.60	23.71	28.04	33.46	40.22	43.58
dynamic normalise rewrite	10.58	14.65	17.51	21.24	26.18	31.67	35.36	40.94	43.70
dynamic length rewrite	10.09	14.19	16.46	20.59	23.40	28.24	33.89	40.83	42.02
dynamic length normalise rewrite	11.00	13.89	17.51	21.93	26.54	30.03	34.24	40.32	41.74
hybrid normalise	14.77	17.51	21.20	24.18	28.08	33.32	36.81	41.50	45.32
hybrid length normalise	13.47	16.77	20.34	23.82	26.25	29.16	33.99	38.71	42.18
hybrid normalise rewrite	15.01	17.91	20.85	23.29	28.71	34.23	38.93	42.76	44.65
hybrid length normalise rewrite	14.32	17.87	19.50	22.23	25.47	32.03	35.69	41.27	42.95

Figure 3: Average reduction in dynamically executed dispatches using varying numbers of superinstructions for each of the schemes

Scheme	8	16	32	64	128	256	512	1024	2048
static	15.35	17.67	21.14	25.16	29.45	35.09	40.93	44.79	49.47
static normalise	15.01	17.67	20.66	24.60	28.65	34.28	40.15	44.27	49.13
static length	12.78	17.71	20.68	24.09	27.84	32.02	39.71	43.41	48.05
static length normalise	12.78	17.66	20.11	23.67	27.02	30.75	37.41	42.93	47.08
static rewrite	15.50	18.46	21.71	25.62	32.55	36.92	42.78	49.30	49.88
static normalise rewrite	15.01	18.39	21.27	25.18	29.99	36.63	42.00	48.92	49.89
static length rewrite	15.26	18.51	21.06	24.48	31.24	37.55	42.70	48.49	49.14
static length normalise rewrite	14.26	17.94	20.14	23.58	29.04	36.49	41.65	48.52	49.21
dynamic	12.35	14.41	16.18	18.95	21.65	23.73	27.58	31.55	36.71
dynamic normalise	12.30	13.14	17.67	21.13	22.63	26.01	28.90	34.69	39.78
dynamic length	10.91	13.42	15.04	16.10	18.29	22.18	25.99	30.46	34.35
dynamic length normalise	10.10	13.14	15.63	16.56	19.41	22.90	27.30	32.80	38.26
dynamic rewrite	11.63	12.92	15.25	18.47	22.10	26.07	30.13	39.14	47.86
dynamic normalise rewrite	11.61	14.64	16.26	18.75	23.43	29.15	33.60	39.54	47.85
dynamic length rewrite	10.05	11.80	14.73	18.16	21.30	24.37	30.60	42.12	46.96
dynamic length normalise rewrite	12.19	13.32	14.67	18.19	22.73	27.70	32.88	42.69	46.87
hybrid normalise	15.01	17.67	20.66	24.60	28.65	34.27	40.18	44.19	48.19
hybrid length normalise	12.78	17.66	20.11	23.67	26.96	30.76	37.41	42.90	47.09
hybrid normalise rewrite	15.01	18.39	21.27	25.18	29.97	36.64	41.71	48.89	49.90
hybrid length normalise rewrite	14.26	17.94	20.14	23.58	29.11	36.51	41.69	48.51	49.19

Figure 4: Average reduction in primitives statically appearing in the program using varying numbers of superinstructions for each of the schemes

number of times they are executed dynamically.

length Sequences are weighted by length. The frequency of the sequence is multiplied by its length during ranking.

normalise Frequencies are normalised across programs. That is, the static (dynamic) frequency of the sequence is divided by the total number of static (dynamic) primitives in the program.

hybrid Sequences are ranked on the sum of their normalised static and dynamic frequencies.

rewrite After the highest ranking superinstruction is chosen, the program is rewritten using the current set of superinstructions. Rankings are then recalculated, based on the new program.

We tested as many combinations of the different schemes as seemed sensible. Figure 3 shows the reduction in dynamically executed dispatches using each of the schemes, and varying the number of superinstructions chosen between 8 and 2048. The numbers presented are average figures across all five programs. It is important that the reduction varied a lot from program to program. For example, with just eight superinstructions the number of dispatches could be reduced for brew by 26.87%, whereas for benchgc the reduction was only 3.7%.

A number of trends are clear from the figures in Fig. 3, and these don't vary from on program to another. First, static measures of frequency consistently outperform dynamic measures. For eight superinstructions, static heuristics reduce the number of dispatches by about one quarter more than dynamic measures. As the number of superinstructions rises, the gap gets smaller, but static measures consistently outperform. Dynamic schemes choose the sequences in the inner loops of the training programs, rather than more generally applicable ones, which might be useful for programs that we have not yet seen.

Another interesting result is that normalising the frequencies doesn't make very much difference to either static or dynamic schemes. When the number of superinstructions becomes very large, it appears to have some benefit, but the result is not consistent.

Surprisingly, weighting sequences by length does not result in more useful superinstructions. Although longer sequences remove more dispatches, it appears that it is much more difficult to match a longer sequences, which offsets the benefit.

Rewriting the training programs between choosing superinstructions gives mixed results. It appears to definitely be a bad idea for dynamic schemes. The problem with rewriting is that it tends to eliminate substrings of long sequences that have been chosen. Dynamic schemes first choose the sequences from the inner loop, and if these are super-strings of useful sequences, rewriting may greatly decrease the priority of these useful short sequences. On the other hand, static schemes benefit from rewriting, since these tend to choose shorter sequences first, and rewriting reduces the priority of other similar strings.

Overall, the best performing scheme is the hybrid one with rewriting. It appears that this successfully combines the advantages of rewriting for static strings with some ability to identify sequences that are likely to execute frequently. The margin over other schemes is small, however. Simple static frequency is remarkably strong, and much simpler to implement.

Figure 4 shows the percentage reduction in static primitives using each combination of the schemes and varying numbers of superinstructions. Perhaps the most interesting result is that the schemes who do best dynamically are also the best statically. In addition to reducing the number of dispatches, we can also expect a significant fall in the size of the interpreted code.

Interestingly, the static reduction in primitives is greater than the dynamic number. We believe that the main reason for this is that a large percentage of the static code consists of Forth words that are part of the GForth system, most of which are invoked on startup. These words are common to all programs, so for these words the training code is actually the same as the real code. In previous work [GEW01] we measured the proportion of static that is executed at least once and belongs to the GForth system and found that it accounts for an average of 65% of static code. These words account for a much smaller proportion (average of 29%) of dynamically executed primitives however, so the dynamic reduction is less.

7 Conclusion

Instruction dispatch is responsible for most of the running time of Forth interpreters, especially on modern pipelined machines. Superinstructions are an important optimisation to reduce the number of instruction dispatches. Many interpreters use superinstructions, but there has been little study of how the sequences which will be turned into superinstructions should be chosen.

In this paper we have examined a number of heuristics for choosing superinstructions for general purpose programs. We found that using the heuristics described in this paper, the number of statically executed dispatches can be reduced by up to an average of 45%. Even using only eight superinstructions, dispatches can be reduced by 15%. Perhaps our most interesting finding is that static measures of sequence frequency are usually better for choosing superinstructions than dynamic measures. We believe that this is because dynamic measures are dominated by a few inner loops, so the sample of code is rather small. In addition, statically frequent sequences are also the most effective at reducing the size of the interpreted program, giving better running time and smaller code size.

Acknowledgments

We would like to thank the anonymous reviewers for their detailed comments which greatly improved the quality of this paper. We are also grateful to Anton Ertl for help with Gforth.

References

[Alm87] T. Almy. Compiling of Forth for performance. *Journal of Forth Applications and Research*, 4(3), 1987.

[BCW90] Timothy Bell, John Cleary, and Ian Witten. *Text Compression*. Prentice Hall, 1990.

[BL75] J. Bruno and T. Lassagne. The generation of optimal code for stack machines. *Journal of the ACM*, 22(3):382–396, 1975.

[DH98a] 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.

[DH98b] K. Driesen and U. Hölzle. The cascaded predictor: Economical and adaptive branch target prediction. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 249–258, Los Alamitos, November 30–December 2 1998. IEEE Computer Society.

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

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

[Ert93] M. Anton Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, Mariánské Lázně (Marienbad), 1993.

[Ert96] M. Anton Ertl. *Implementation of Stack-Based Languages on Register Machines*. PhD thesis, Technische Universität Wien, Austria, 1996.

[GEW01] David Gregg, M. Anton Ertl, and John Waldron. The common case in Forth programs. In *EuroForth 2001 Conference Proceedings*, pages 63–70, 2001.

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

[HP90] John Hennessy and David Patterson. *Computer architecture: A quantitative approach*. Morgan Kaufmann Publishers, 1990.

[Hug82] R. J. M. Hughes. Super-combinators. In *Conference Record of the 1980 LISP Conference, Stanford, CA*, pages 1–11, New York, 1982. ACM.

[KK98] J. Kalamatianos and D. R. Kaeli. Predicting indirect branches via data compression. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 272–284, Los Alamitos, November 30–December 2 1998. IEEE Computer Society.

[Luc00] Steven Lucco. Split-stream dictionary compression. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 27–34, 2000.

[Piu98] Ian Piumarta. Optimizing direct threaded code by selective inlining. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, June 1998.

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

[Ros86] Anthony Rose. Design of a fast 68000-based subroutine-threaded Forth with inline code & an optimizer. *Journal of Forth Application and Research*, 4(2):285–288, 1986. 1986 Rochester Forth Conference.

[Sch92] Udo Schütz. Optimierung von Fadencode. In *FORTH-Tagung*, Rostock, 1992. Forth Gesellschaft e.V.