The Common Case in Forth Programs

David Gregg *
Trinity College Dublin
David.Gregg@cs.tcd.ie

M. Anton Ertl
Technische Universität Wien
anton@complang.tuwien.ac.at

John Waldron
Trinity College Dublin
John.Waldron@cs.tcd.ie

Abstract

Identifying common features in Forth programs is important for those designing Forth machines and optimisers. In this paper we measure the behaviour of six large Forth programs and four small ones. We look at the ratio of user to system code, basic block lengths, common instructions, and common sequences of instructions. Our most important finding is that for most large programs, many (38.4%–47.6% statically and 21.8%–40.9% dynamically) basic blocks consist of only a single instruction, which hinders optimisation. We also show static measures of frequent instructions and sequences of instructions are more consistent across programs, and may be a better predictor of the behaviour of other programs than dynamic measures.

1 Motivation

Perhaps the most important principle of modern computer architecture is to make the common case fast [HP96]. In deciding where to devote resources, favour the frequent case over the infrequent one. This is the idea behind RISC processors, virtual memory, caches, and speculative execution. Today, general purpose processors and their compilers are primarily designed to run integer C code, of the type found in SPEC benchmarks, while supercomputers are designed to deal with large array and floating point computations. The design tradeoffs for any computing machine depend on the relative frequency of different types of computations. Thus, the designer of a Forth interpreter (a processor is simply a hardware implementation of an interpreter) must consider the common cases in typical Forth programs.

In this paper we examine the behaviour of a number of Forth programs. We look at the most commonly executed instructions both statically and dynamically. We also examine the break-down of executed primitives between user code and system code. We look at important metrics, such as basic block length. And we find the most commonly occurring sequences of instructions within a basic block, and consider how well observations about one program are applicable to other Forth programs.

We are not the first to make measurements such as these. As we show in section 2 a number of authors have examined the behaviour of Forth programs. The work in this paper differs from all existing work in at least one of three ways, however. Most importantly, we focus real programs of several hundred to several thousand lines rather than small benchmarks. We do measure a number of microbenchmarks as well, but only to show that their behaviour is very different from larger programs. Secondly, we measure a relatively large number of programs (ten in total), written by several different authors, which helps us to avoid biases from unrepresentative individual programs. Finally, our results are presented in a paper that can be cited by other researchers with confidence that the measurements have been peer reviewed.

The remainder of this paper is organised as follows. We first describe existing work on the behaviour of Forth programs (section 2). In section 3 we describe six large benchmarks and four small which we use for our measurements. In section 4 we describe our method of collecting data, and presents measurements of the benchmarks. Finally, in section 5 we draw conclusions.

2 Related Work

In the process of designing Forth machines, a number of previous authors have examined the behaviour of Forth programs. Hayes et al [HFWZ87] measured the dynamic frequency of a various instructions for two large programs and one small one. They found that calls and returns were very frequent in the large program, but less frequent in the small one.

Koopman [Koo89] measured the execution frequency of Forth instructions in six small benchmark programs. In all benchmarks, `CALL` and `EXIT` instructions were the most common (at about 12% each). The frequency of other instructions varied

*Supported by Enterprise Ireland International Collaboration Programme, Project IC/2001/024

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: EuroForth 2001 Conference Proceedings, pages 63-70

more widely across benchmarks. This is not surprising, since what is essentially being measured is which instructions are in the inner loop of these small programs.

In previous unpublished work Ertl measured the frequency of instructions, and sequences of instruction in three large Forth programs. Again calls and returns were the most common instructions, but there was more consistency in the other frequent instructions across programs. The raw data from these measurements can be found at `http://www.complang.tuwien.ac.at/forth/peep/`.

3 Benchmarks

This section describes the benchmarks that we use to measure the behaviour of typical Forth programs. We have chosen four small benchmarks and six larger ones. We believe that the results from the large benchmarks are generally representative of real programs. We examine the small programs for comparability with existing work, and to show that the behaviour of small and large programs can be very different.

The small benchmarks are those that come with the publically available Gforth system. Lines of source code include blank lines and comments, and are shown in Fig. 1. The programs are:

sieve Sieve of Eratosthenes program which computes all primes between 2 and 8190.

bubble Bubble sort an array of 6000 elements.

matrix Multiplication of 200 by 200 matrices.

fib A naive recursive computation of the first 34 Fibonacci numbers.

The larger benchmarks are real programs from a variety of application areas. We have been careful to choose programs from a number of different authors to avoid measuring just a single programming style. The larger 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 output a C source interpreter for the instructions.

grey 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.

pentomino A puzzle solving problem that tries to fit twelve pentominos (shapes constructed from five squares) together. Pentomino includes a code generator that produce large numbers of very similar new routines at run time. Thus, the run time code size is much larger than would be suggested by the 511 lines of source.

4 The measurements

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. This allows the Forth engine to be simple and portable, while allowing the programmer to add new words to the system without recompiling existing code.

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 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 next section shows the result of these measurements.

4.1 User versus system code

When a Forth program is run, both system and user-defined words are executed. An important question is the relative proportion of the two that make up the running time of the program. For example, it is well known that general purpose Java programs spend much of their time executing code from the standard Java libraries [Wal99], and that most programs with a graphical user interface spend much of their time in graphics libraries.

System code in Gforth is executed for two main reasons. First, much of Gforth is written in Forth. For example, when a program is loaded the text interpreter (which is written in Forth) interprets the commands and compiles colon definitions. Secondly, the Forth system provides many words which can be invoked by user programs. Many of these system words are not primitives, so they result in Forth code being called. The question is how much of the code in a typical run of a program is system code.

Figure 1 shows the static and dynamic proportion of system code for each of the benchmarks. For the small benchmarks the pattern is clear. Statically, almost all of the code is system code. The Gforth code to startup and load the benchmark consists of more than 2000 primitives, which is far larger than any of the small benchmarks. However, almost 100% of the dynamically executed instructions are in user code, since these small programs spend almost all their time in a simple, tight loop.

The results for large programs are quite different, demonstrating the danger of assuming that large programs behave the same as small ones. Not surprisingly, system code makes up a small proportion of the static instructions, since the programs are larger. More interestingly, the large programs spend a large proportion of their time in system code. This can partly be explained by the text interpreter compiling more user code, much of which will execute only a small number of times. The rest can be accounted for by the user programs using system words. This is especially clear in `gray` and `prims2x`, which use string manipulation words for their parsing and output.

Generally, Forth programs spend relatively little time in system code, so providing an efficient Forth execution engine is more important than optimising system words. Nonetheless, system code is not insignificant, so its efficiency cannot be entirely neglected.

4.2 Basic block lengths

A basic block is a sequence of instructions where no instruction but the last is a branch, and no instruction but the first is the target of a branch. Basic blocks are important because there is no branching within them, which greatly simplifies optimisations. Figure 2 shows the percentage frequency of various basic block lengths in the static code. Perhaps the most remarkable result is the very large number of basic blocks consisting of only a single instruction (31.6% – 47.6%). The reason for this is the very large numbers of calls and returns in Forth code. In addition to the normal control flow instructions, Forth executes a call or return about one instruction in every four (see section 4.3). Also notable are the large number of basic blocks of length 2 or 3. Based on the static frequencies, it appears that optimisations attempt to improve the code in basic blocks are unlikely to be very successful; there are too few instructions to optimise.

Figure 3 shows the basic block length distribution weighted by execution frequency. For the large benchmarks, the proportion of very short basic blocks is smaller than for the static figures, but still high. For the smaller benchmarks, there are a handfull of frequent lengths and almost no others. The small benchmarks consist primarily of a small loop which executes many times. The frequent blocks are those that appear in that loop. Furthermore, most of the small benchmarks rarely execute basic blocks of length one. Again, they spend most of their time in small loops rather than in calls and returns.

This last finding explains a result from our work on interpreter optimisation which originally puzzled us. We have implemented an optimisation that combines frequent sequences of instructions into a single "superinstruction" [Pro95]. Our initial results show that the optimisation gives excellent speedups for small programs, but is less successful for large ones. The reason is now clear. Large Forth programs contain many very short basic blocks, so the opportunities for combining are limited.

There are two exceptions to our general observation about small and large programs. The benchmark `fib` consists of a small, heavily recursive function, rather than a tight loop, so it contains many calls. Secondly, `pentomino`, spends most of its time in code generated at run time which contains a basic block with six instructions. These programs show the importance of looking at a variety of benchmarks, rather than depending on a small number of examples.

4.3 Frequencies of primitives

The most important measure of the behaviour of a Forth program is which primitives are executed. The Forth engine should be optimised for the most frequently occurring instructions. Figure 5 shows the most frequently statically appearing instructions for our large benchmarks. The most remarkable feature is the large number of `call` and `;s` (which implements the `exit` word for returning from calls) instructions. Statically, they account

Benchmark	source lines	static instrs.	static %	dynam. instrs.	dynamic %
prims2x	1258	6,314	43.27	18,319,272	51.33
gray	1458	4,792	37.92	4,833,582	25.57
brew	7627	6,078	43.43	1,312,698,495	93.94
brainless	3755	11,430	72.34	859,030,440	87.73
benchgc	1479	4,105	26.35	559,786,379	91.56
pentomino	511	7,951	63.92	746,097,406	99.26
siev	20	2,325	2.54	150,829,255	99.79
matrix	55	2,283	4.38	139,192,159	99.75
fib	10	2,075	0.96	194,052,085	99.86
bubble	74	2,426	3.59	170,931,912	99.72

Figure 1: User code as a percentage of total code

Benchmark	1	2	3	4	5	6	7	8	9	10	11	12	13	14	≥15
prims2x	47.1	21.2	12.1	5.5	4.5	3.0	2.0	1.5	0.6	0.7	0.5	0.2	0.2	0.0	0.6
gray	47.6	22.0	13.3	5.6	4.1	2.1	1.6	1.3	0.6	0.6	0.1	0.2	0.1	0.1	0.5
brew	39.0	20.1	15.4	9.4	6.0	3.0	2.2	1.7	1.0	0.8	0.2	0.3	0.2	0.3	0.5
brainless	45.3	17.0	12.1	6.4	5.0	3.3	7.8	1.3	1.0	0.7	0.5	0.7	0.3	0.3	1.1
benchgc	38.4	18.5	16.0	8.5	6.4	3.8	2.5	2.1	1.1	0.9	0.2	0.5	0.1	0.1	0.7
pentomino	31.6	13.2	17.3	6.5	5.3	9.6	2.0	1.6	1.0	0.7	0.5	0.4	0.1	0.1	10.0
siev	34.2	19.6	16.0	8.4	7.0	4.4	3.0	2.6	1.5	1.1	0.1	0.5	0.1	0.3	1.1
matrix	33.7	19.4	15.8	8.2	7.6	4.6	3.1	2.4	1.5	0.9	0.1	0.7	0.3	0.3	1.3
fib	34.2	19.8	16.0	7.9	7.3	4.2	3.0	2.6	1.7	1.1	0.2	0.5	0.2	0.3	0.9
bubble	36.4	18.8	15.3	8.2	7.2	4.1	3.0	2.4	1.4	0.9	0.1	0.6	0.1	0.3	1.0

Figure 2: Percentage distribution of static basic block lengths

Benchmark	1	2	3	4	5	6	7	8	9	10	11	12	13	14	≥15
prims2x	40.9	15.2	11.5	6.9	4.5	10.6	2.3	3.5	0.6	1.8	0.1	0.5	1.3	0.0	0.0
gray	32.3	27.3	18.0	6.1	5.5	4.2	2.3	1.6	1.0	0.6	0.0	1.0	0.0	0.0	0.0
brew	27.4	16.4	11.6	12.3	12.5	6.2	8.6	1.0	0.5	1.8	0.3	0.1	0.0	1.3	0.0
brainless	31.7	17.0	11.8	8.6	5.9	5.5	5.6	3.4	1.3	1.2	3.4	0.7	0.5	1.3	2.4
benchgc	21.8	17.9	14.6	13.7	5.3	18.2	1.9	1.0	3.4	1.8	0.0	0.3	0.0	0.0	0.0
pentomino	0.8	0.3	12.1	13.1	2.7	50.6	13.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	7.2
siev	3.1	1.9	48.3	1.9	39.6	0.0	0.0	5.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
matrix	1.1	0.2	0.6	0.2	1.0	0.6	0.1	0.0	0.0	0.0	0.0	0.9	0.9	0.0	94.2
fib	37.4	0.0	25.0	37.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
bubble	0.1	0.1	40.0	40.0	19.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 3: Percentage distribution of dynamic basic block lengths

for about one quarter of all instructions in most benchmarks. Given that calls and returns do no useful computation, they seem a source of unnecessary inefficiency. Forth programs could benefit considerably from procedure inlining. Not only would it reduce the number of calls and returns; it would also increase the length of basic blocks, allowing other optimisations to be more effective.

Another notable feature is the large number of `lit`, `@` and `useraddr` instructions. As with most machines, at least as much of the time is spent data shuffling data to where it can be operated up (in this case the stack) as is spent operating on it. Stack manipulation operations (such as `dup`, `swap`, `over`, `>r`, `r>`) are also common. Such operations may be a source of optimisation if the stack can be arranged in a more efficient way to reduce the number of manipulations. Procedure inlining might make this more effective if it exposes more stack manipulation operations simultaneously to such an optimisation phase.

Figure 5 shows the twenty dynamically most frequent instructions for the six large benchmarks. Calls and returns are slightly less frequent than in the static measurements, which explains why dynamic basic block lengths tend to be longer than

prims2x		gray		brew		brainless		benchgc		pentomino	
call	21.73	call	22.43	lit	19.00	lit	23.61	lit	17.44	lit	30.81
lit	18.21	lit	19.22	call	18.28	call	17.61	call	16.10	+	9.94
@	9.44	;s	10.22	@	11.17	@	15.32	@	9.67	c!	9.58
;s	9.15	@	9.01	;s	8.35	;s	7.47	;s	9.43	over	8.39
?branch	3.64	?branch	4.01	?branch	4.33	?branch	3.50	dup	4.80	call	6.45
+	2.77	dup	2.80	dup	3.95	dup	2.96	?branch	3.75	@	4.63
dup	2.77	!	2.63	!	2.71	!	2.93	!	3.36	;s	3.74
!	2.39	execute	2.39	execute	2.04	+	2.76	swap	2.14	?branch	3.41
execute	2.33	>r	1.67	>r	1.76	swap	1.37	>r	2.12	dup	3.37
swap	1.35	+	1.63	+	1.69	over	1.27	execute	2.00	c@	2.23
>r	1.46	swap	1.57	swap	1.58	and	1.16	r>	1.83	execute	1.62
branch	1.43	branch	1.57	r>	1.53	drop	1.09	drop	1.80	0=	1.51
drop	1.35	r>	1.44	drop	1.35	branch	1.06	+	1.80	!	1.28
i	1.27	drop	1.29	branch	1.28	cells	1.05	over	1.77	+!	0.97
r>	1.19	over	1.25	0=	1.17	=	1.03	useraddr	1.36	>r	0.89
over	1.17	useraddr	1.19	i	1.17	execute	0.01	branch	1.19	swap	0.74
useraddr	1.11	2dup	0.86	useraddr	1.14	i	1.00	2dup	1.12	useraddr	0.74
0=	0.95	0=	0.79	over	1.12	>r	0.91	0=	1.09	r>	0.74
2dup	0.84	-	0.73	+!	0.79	r>	0.86	-	1.09	drop	0.65
2@	0.73	i	0.73	=	0.79	or	0.85	i	1.08	2dup	0.49

Figure 4: Statically frequent instructions for large benchmarks

static ones (see section 4.2).

Generally, the similarity between the lists of static and dynamic instructions is remarkable. The same instructions that appear statically frequently are also strongly represented in the inner loops of the large programs. Some instructions which are specific to these programs (such as `rshift` in benchgc and `mod` in brew) do not appear in the static instructions, but in general the lists are much the same. If one had to identify the most important instructions the static and dynamic lists could be used almost interchangeably.

An exception to this is the `(loop)` instruction (in brainless, benchgc and pentomino). Not surprisingly, this instruction appears only rarely in static code, but is executed many times dynamically, because it is used for looping. This suggests that while static measures will identify most of the most important instructions, it is also important to measure programs dynamically.

Pentomino's dynamic behaviour is very different to that of the other programs. Clearly the dynamic figures for pentomino are dominated by an inner loop which executes many times. Examining the list, one can see which instructions appear in its inner loops and how many times they appear. The top ten instructions account for almost all executions. While these instructions are mostly the ones common in other programs, the presence of `c!` so high in the list shows that the machine designer must be prepared for the occasional program which contains an unusual instruction in its inner loop.

In addition to the instruction frequencies for the large benchmarks, we made the same measurements for the four small benchmarks. We found that the statically frequent instructions are very similar across all four small benchmarks, and also similar to the static frequencies for the large benchmarks. The reason is that most of the statically occurring code in the small benchmarks is system code (section 4.1) which is the same for all four benchmarks, and also occurs in the large benchmarks. The dynamic frequencies are even more extreme than for pentomino, with just a handfull of primitives accounting for almost all executed instructions.

4.4 Frequencies of sequences

A number of optimisations can be applied to Forth code which rely on identifying frequent sequences of instructions. We have already mentioned "superinstructions" (see section 4.2). Another example is to base predictors for code compression schemes on the frequency of instruction sequences [EEF+97].

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

prims2x		gray		brew		brainless		benchgc		pentomino	
lit	15.26	;s	15.08	lit	22.31	lit	19.77	lit	8.20	lit	24.57
@	12.79	@	14.88	@	19.13	@	15.79	;s	8.05	+	12.07
;s	12.18	call	14.07	;s	11.18	;s	8.86	call	7.94	?branch	12.05
call	11.13	lit	12.17	call	10.00	call	8.38	@	7.09	dup	11.03
?branch	4.78	?branch	3.89	+	7.28	+	7.80	and	6.14	c@	10.24
swap	3.89	+	3.82	?branch	3.94	?branch	4.63	dup	5.90	0=	8.89
+	3.34	useraddr	3.18	!	3.52	cells	4.18	over	5.67	c!	5.93
0=	2.76	dup	2.44	dup	3.11	dup	4.11	i	5.47	over	4.96
dup	2.31	!	2.41	+!	1.61	and	2.63	(loop)	4.75	1+	3.64
r>	1.77	over	2.13	swap	1.32	=	2.43	=	4.55	@	2.27
>r	1.77	swap	1.73	execute	1.30	over	2.17	c@	4.40	>	1.81
=	1.67	r>	1.71	over	1.14	!	2.15	?branch	4.23	+!	0.99
c@	1.60	>r	1.69	=	1.12	swap	1.47	swap	3.48	;s	0.44
and	1.41	execute	1.49	cells	1.11	?dup	1.46	>r	3.00	execute	0.37
>l	1.26	0=	1.32	mod	0.98	i	1.04	+	2.72	drop	0.36
useraddr	1.25	and	1.12	i	0.96	*	0.99	r>	2.11	call	0.07
count	1.10	cell+	1.11	drop	0.93	drop	0.97	-	1.35	(loop)	0.04
cells	1.06	=	1.04	branch	0.82	(loop)	0.84	rshift	1.33	i	0.02
!	1.04	2dup	0.97	>r	0.78	or	0.80	0=	1.26	emit-file	0.02
over	1.03	?dup	0.84	r>	0.71	useraddr	0.72	drop	1.25	*	0.02

Figure 5: Dynamically frequent instructions for large benchmarks

block. For this reason, we used a more sophisticated scheme to measure only those sequences which appear within basic block boundaries.

Figure 7 shows the twenty-five statically most common sequences for each of the large benchmarks. As expected, the most common sequences are short and consist of combinations of the most frequent instructions. The combinations are not random, however. For example, it is interesting to note the frequency with which a constant is pushed onto the stack, followed by a call (`lit call`). It is also common to store a value just before returning from a call (`! ;s`). Another interesting feature is that although calls, returns and branches form a very large percentage of statically appearing instructions, they are much less frequent in sequences. The reason is that they terminate basic blocks, so they can only appear at the end of a sequence.

The results for two programs stand out from the others. First, brainless, which is the largest of our benchmarks, contains 193 instances of a single long sequence (`lit @ lit @ lit @ call`). This sequence, and all substrings of are responsible for most of its entries. Secondly, pentomino generates large amounts of very similar code at run time. Among this code are 496 instances of the sequence `lit over lit + c! lit over lit + c!`, which dominate the results. Clearly the Forth goal of code factoring is not working well for either of these programs.

Figure 6 shows the twenty-five dynamically most frequent sequences of instructions. Many of the sequences also appear on the static lists (e.g. `lit @`, `lit call`). However, many more of the sequences are clearly program specific. For example, many of the most common sequences consist of the basic block in the most frequent inner loop, and all substrings of that block. There is much more variation among the dynamic sequences for the different programs than for the static ones. This suggests that if one wishes to identify which sequences are likely to be important for Forth programs in general, it may be better to use static sequences from a basket of programs rather than dynamic ones.

5 Conclusion

Identifying common features in Forth programs is important for those designing Forth machines and optimisers. In this paper we have measured the behaviour of six large Forth programs and four small ones. Forth programs spend most of their time in user rather than system code, suggesting that effort should be spent on improving Forth engines rather than tuning libraries. Large programs execute large numbers of call and return instructions, which results in very short basic blocks, which hinder optimisation. Procedure inlining may reduce this problem. We examined the static and dynamic frequency of instructions and found that a handfull of frequent VM instructions account for most execution. In addition we also looked at the most frequently appearing sequences of instructions

prims2x	gray	brew	brainless	benchgc	pentomino
lit @	lit @	lit @	lit @	over i	lit +
@ lit	@ ;s	@ +	lit +	= and	0= ?branch
lit call	lit call	lit @ +	cells lit	over i c@ = and (loop)	c@ 0= ?branch
= ?branch	useraddr @	@ lit	cells lit +	over i c@ = and	c@ 0=
lit @ lit	+ ;s	+ ;s	@ lit	over i c@ =	dup lit
lit @ c@	! ;s	@ + ;s	+ @	over i c@	+ c@
@ c@	useraddr @ ;s	lit @ + ;s	lit + @	i c@ = and (loop)	lit + c@
@ lit @	@ +	@ lit @	cells lit + @	i c@ = and	dup lit +
lit @ lit @	@ lit	@ lit @ +	lit @ =	i c@ =	lit + c@ 0= ?branch
@ ;s	@ + ;s	@ lit @ + ;s	@ =	i c@	lit + c@ 0=
@ =	@ call	lit !	lit @ and	c@ = and (loop)	dup lit + c@ 0= ?branch
lit @ = ?branch	@ execute	lit @ lit	@ and	c@ = and	dup lit + c@ 0=
lit @ =	lit @ execute	lit @ lit @	+ dup	c@ =	dup lit + c@
@ = ?branch	= ?branch	lit call	@ lit @	and (loop)	+ c@ 0= ?branch
useraddr @	over = ?branch	lit @ lit @ +	dup lit	= and (loop)	+ c@ 0=
@ and	over =	! ;s	lit !	lit @	lit over
0= ?branch	@ useraddr	+ lit	?dup ?branch	dup lit	over lit
0= 0=	@ useraddr @	lit ! ;s	@ ;s	lit call	c! lit
swap @	@ @	+ lit @	lit @ lit	+ ;s	+ c!
lit lit	?dup ?branch	dup lit	and lit	lit and	lit + c!
and 0=	cell+ swap	lit @ lit @ + ;s	lit call	@ +	over lit + c!
@ and 0=	useraddr @ + ;s	@ + lit	@ +	@ + ;s	over lit +
lit swap	@ useraddr @ + ;s	lit @ + lit	@ lit @ and	dup lit and	lit over lit + c!
+ swap	useraddr @ +	lit @ + lit @	@ ?dup	dup @	lit over lit +
swap ;s	@ useraddr @ +	@ + lit @	@ ?dup ?branch	@ dup	lit over lit

Figure 6: Dynamically frequent sequences of instructions for large benchmarks

and found that the same sequences tend to appear statically frequently in all large programs. On the other hand, there is considerable variation in the frequency of dynamically appearing sequences. In many cases the dynamic figures simply measure the contents of the inner loop, suggesting that it would be better to generalise from the static frequencies. These measurements are useful for identifying future optimisation opportunities for Forth machines.

References

[EEF+97] Jens Ernst, William Evans, Christopher W. Fraser, Steven Lucco, and Todd A. Proebsting. Code compression. In *SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 358–365, 1997.

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

[HFWZ87] John R. Hayes, Martin E. Fraeman, Robert L. Williams, and Thomas Zaremba. An architecture for the direct execution of the Forth programming language. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pages 42–48, 1987.

[HP96] John L. Hennessy and David A. Patterson. *Computer architecture — a quantitative approach, Second Edition.* Mac Graw-Hill, 1996.

[Koo89] Philip J. Koopman, Jr. *Stack Computers.* Ellis Horwood Limited, 1989.

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

[Wal99] John Waldron. Dynamic bytecode usage by object oriented java programs. In *Proceedings of the Technology of Object-Oriented Languages and Systems 29th International Conference and Exhibition*, Nancy, France, June 7-10 1999.

prims2x	gray	brew	brainless	benchgc	pentomino
lit @	lit @	lit @	lit @	lit @	lit +
lit call	lit call	lit call	@ lit	lit call	lit over
@ execute	@ execute	@ lit	lit @ lit	lit @ execute	+ c!
lit @ execute	lit @ execute	@ execute	@ lit @	@ execute	c! lit
lit +	@ lit	lit @ execute	lit @ lit @	lit lit	over lit
@ lit	@ call	lit @ lit	@ call	dup lit	lit + c!
lit @ lit	lit !	lit lit	lit call	lit !	over lit + c!
@ call	lit @ lit	lit !	lit @ call	@ lit	over lit +
+ @	lit lit	@ call	lit !	@ call	lit over lit + c!
lit lit	! lit	dup lit	lit @ lit @ lit	! lit	lit over lit +
lit + @	lit @ call	@ lit @	@ lit @ lit	! ;s	lit over lit
@ lit +	! ;s	! lit	lit @ lit @ call	lit @ call	+ c! lit
lit !	dup lit	0= ?branch	@ lit @ call	lit @ lit	lit + c! lit
! lit	@ ;s	! ;s	lit @ lit @ lit @	dup call	over lit + c! lit
lit @ lit +	lit +	lit @ call	@ lit @ lit @	0= ?branch	lit over lit + c! lit
dup lit	lit @ ;s	dup call	lit @ lit @ lit @ call	lit +	over lit + c! lit over lit + c!
! ;s	useraddr @	lit @ lit @	@ lit @ lit @ call	useraddr @	over lit + c! lit over lit +
lit @ call	0= ?branch	= ?branch	lit +	lit ! lit	over lit + c! lit over lit
0= ?branch	lit ! lit	@ +	lit lit	dup @	over lit + c! lit over
useraddr @	useraddr !	i call	dup lit	dup ?branch	+ c! lit over lit +
= ?branch	dup ?branch	@ ?branch	! lit	useraddr !	c! lit over lit
useraddr !	= ?branch	useraddr @	@ execute	drop ;s	c! lit over lit +
@ ;s	dup call	dup @	lit @ execute	lit lit @	+ c! lit over lit + c!
dup call	@ lit @	dup ?branch	lit ! lit	@ dup	c! lit over lit + c!
dup ?branch	drop ;s	lit +	@ ?branch	= ?branch	lit + c! lit over

Figure 7: Statically frequent sequences of instructions for large benchmarks