RAFTS for Basic Blocks: A progress report on Forth Native Code Compilation*

M. Anton Ertl Christian Pirker

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien, Austria
{anton,pirky}@mips.complang.tuwien.ac.at
Tel.: (+43-1) 58801 4474
Fax.: (+43-1) 505 78 38

Abstract. RAFTS is a framework for applying state of the art compiler technology to the compilation of stack-based languages, in particular Forth. This paper describes our experiences and findings in implementing the basic block (straight-line code) part of RAFTS; it also presents empirical results: the basic block part is the simplest part of RAFTS, but without the other parts it is hardly better than existing techniques, because in Forth basic blocks are very short (there are few basic blocks with more than two useful instructions).

1 Introduction

Traditionally, Forth is implemented using a threaded code interpreter [Tin81]. This keeps the compiler extremely simple: a word is compiled by appending its address to the current definition. Performance bottlenecks are removed by manually recoding the critical pieces of code in assembly language. While this may be appropriate in embedded systems programming, it does not meet the portability requirements for general-purpose computing.

Another popular avenue to efficiency is the creation of special hardware, which has led to many research-prototype and several commercially available CPUs that are designed to run Forth efficiently [Koo89, HFWZ87]. The code for these stack machines is generated by simple, peephole-optimizing compilers.

Recent research [Ung87, CU89, KB92] proves that unconventional languages can be implemented efficiently on mainstream hardware using aggressive compiler techniques and that the gains of specialized architectures are usually offset by the better device technology available to widely-used RISCs.

RAFTS[1] is a framework for applying such aggressive compiler techniques to stack-based languages, in particular Forth. The special needs of Forth are fast stack manipulation and fast procedure calls. The RAFTS framework has been described in [Ert92]; an improved version can be found in [Ert96]. The first version of the implementation has been described in [Pir95].

This paper describes our experiences with the basic block part of RAFTS, i.e., the part that deals with straight-line code. Section 3 recapitulates the basic block part of the RAFTS framework. Section 4 describes the experiences and findings in implementing that part. Section 5 and Section 6 provide some empirical data about the effectiveness (or rather, ineffectiveness) of using the basic block part of RAFTS alone.

2 Related Work

BNF-based parsers can be used to transform postfix code into trees, i.e., data flow graphs. But parsers cannot process stack manipulation words and multiple stacks, and the method of Section 3.1 is simpler than parsing.

Stack-based languages have been used as intermediate code for compilers, e.g., [TvSKS83]. During code generation, these compilers face problems similar to Forth native code generation. However, they do not have to handle multiple stacks or stack manipulation words like `SWAP` and they do not use the stacks across basic block boundaries. Instead, they put a much higher emphasis on local and global variables. The code generator of the Amsterdam Compiler Kit (ACK) resembles the method of Section 3.1 in its use of the stack, but it performs all tasks of code generation at the same time [TvSKS83]. Everything else in ACK is different: in contrast to RAFTS, which does not do anything at the Forth level, ACK optimizes as much as possible at the postfix code level, because its main goal is machine-independence.

HP and Tandem have used binary-to-binary translators to move existing programs in executable form from stack architectures to (RISC) register architectures. [AS92] describes Tandem's binary

* This work is supported by the FWF, research project P11231-MAT.

[1] RISCs Are Faster Than Stack machines.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: EuroForth '96 Conference Proceedings

translator, but gives few details about the stack-to-register conversion and is somewhat machine-specific in these details.

There are several Forth compilers that produce native machine code [Ros86, Alm86, Pay91]. Usually they start by generating subroutine-threaded code, then they inline small subroutines, and peephole-optimize the resulting code to remove redundant pushes and pops. One nice property of such systems is that they can be easy to retarget. The retargetter just has to replace the native code for the primitives and the tables of the peephole optimizer; the compiler proper remains untouched. On these systems, much stack manipulation overhead remains. Rose [Ros86] reports that "the resulting machine code is a factor of two or three slower than the equivalent code written in machine language, due mainly to the use of the stack rather than registers"; our measurements confirm this (see Section 5).

However, some compilers are pretty sophisticated at reducing stack manipulation overhead. JForth V3.0 can keep up to five values in registers and optimizes words like `SWAP` and `ROT` completely away. This optimization is only performed on sequences of certain primitives. Almy's non-interactive (i.e., batch) CFORTH compiler keeps up to two values in registers and reduces stack pointer manipulations [Alm86]. It keeps values in registers across basic block boundaries. However, the lack of global register allocation in his compiler would result in a lot of register shuffling at basic block boundaries, if the compiler used more than two registers. [Sto88, Kna93] show how a compiler using such techniques could be implemented: there is a compilation word for each word to be compiled that performs a case analysis of the preceding code; apart from being error prone, this method is hard to extend to global optimizations (e.g., global register allocation).

Koopman did similar research in the opposite direction: his C compiler keeps variables on the stack in order to execute C efficiently on stack machines [Koo92].

3 RAFTS on Basic Blocks

A basic block is a piece of straight-line code, i.e., it can be entered only at the beginning and left only at the end. At the Forth level, basic blocks consist only of primitives like `+` and `!`, of literals, constants and variables, and of stack manipulation words like `SWAP` and `R@`. Words that change the control flow (e.g., `IF` or calls of high-level words) or are targets of control-flow changes (`BEGIN`, `THEN`) delimit basic blocks.

3.1 Data flow graphs

The data flow graph (DFG) is the basic data structure of several state-of-the-art compiler algorithms[2], and it can be extended and converted to data structures for other algorithms with standard methods.

The compiler builds the data flow graph by symbolically executing the Forth words in the basic block: it maintains the stacks just as an interpreter would during execution. Stack manipulation words are compiled by simply executing them. When one of the other words is compiled, the compiler pops and pushes as many items as the word would during execution. But the word is not executed. Instead, the compiler builds a data flow graph node, which contains a token for the word (to indicate the kind of node) and the operands taken from the stack. A pointer to this data structure is pushed on the stack instead of the result. In this way, the compiler builds a data flow graph for the basic block (see Fig. 1).

3.2 Code generation

The other three steps of the conversion of a basic block from Forth to native code are instruction selection, instruction scheduling and register allocation. They have been described extensively in the literature, so here they will not be explained in-depth.

Instruction selection combines the operators in the data flow graph into legal instructions of the target machine, transforming the operator DAG into an instruction DAG (see Fig. 2).

All referenced stack items now reside in pseudo-registers (px in Fig. 2), of which an unlimited number can be used; all stack accesses within a basic block have been eliminated. **Register allocation** replaces the pseudo-registers with real registers and spills excess pseudo-registers to memory. Edges in the instruction DAG correspond directly to pseudo-registers.

Instruction scheduling orders the instructions, i.e., it transforms the instruction DAG into a list (see Fig. 2). The data flow graph (as constructed above) contains only data flow dependences through the stack, it does not contain data dependences through memory; in other words, it is not a full data dependence graph. Therefore the compiler also keeps track of memory accesses during the symbolic execution: it remembers the last store and introduces dependences from the store to

[2] For basic blocks the graph is a directed acyclic graph (DAG); the data flow graph is so important that [ASU86] refers to it simply as DAG. Many compiler textbooks draw it up-side-down (just like they draw trees), with the data flowing upwards.

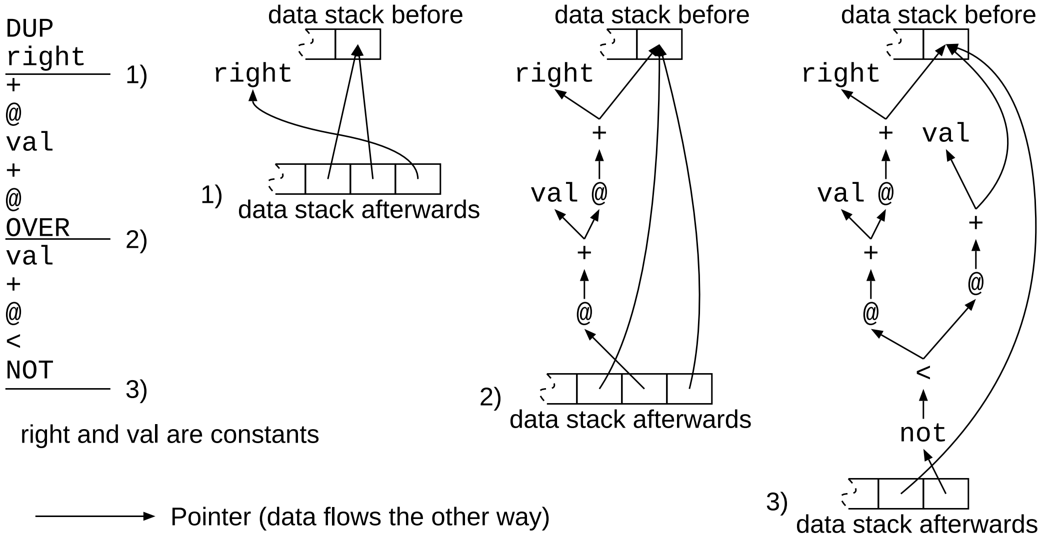

Fig. 1. Building the data flow graph (three snapshots)

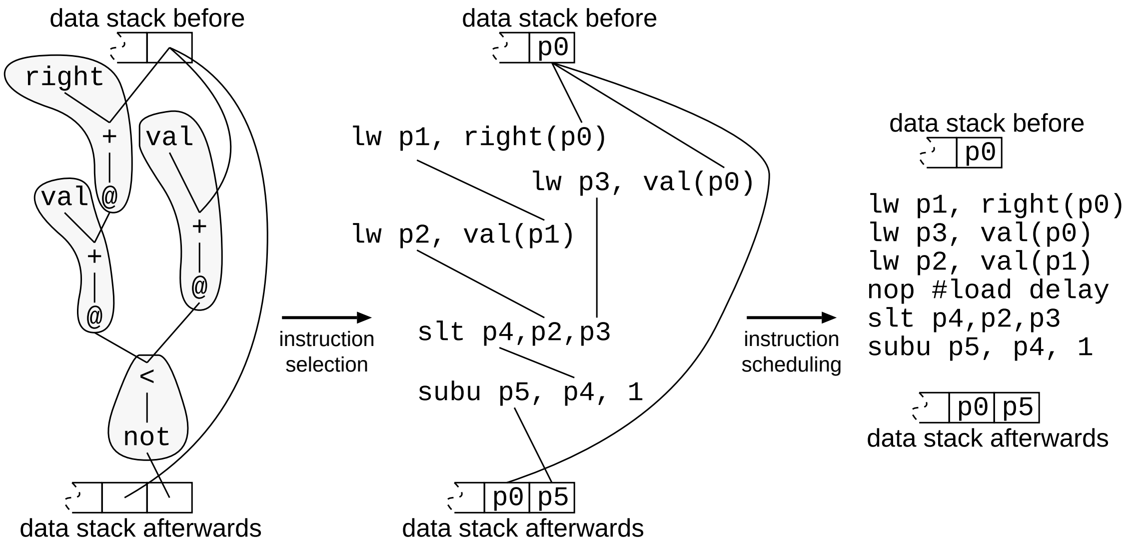

Fig. 2. Instruction selection and scheduling (for the MIPS architecture)

subsequent fetches and the next store; it also keeps a list of fetches since the last store and introduces dependences from the fetches to the next store.[3] Any topological ordering of the data dependence graph constitutes a valid schedule.

4 Implementation

The RAFTS implementation currently produces code for the MIPS architecture. It runs on top of the Gforth threaded code system [Ert93] and is well integrated with it.

[3] This method assumes that all memory accesses are aliased, which is safe, but may cost performance.

The examples are written in MIPS assembly code: machine register `n` is denoted by `$n`, the destination operand of an instruction is usally the left-most register, and comments start with `#`.

The data stack pointer is in the machine register `%sp` (`$17`), the return stack pointer in `%rp` (`$18`).

4.1 Structure of the compiler

When primitives are compiled (i.e., their compilation semantics are performed, e.g., by the text interpreter in compile state), they build the data flow graph of the basic block.

Words that end basic blocks (e.g., control flow words) generate the machine code for the current basic block as part of their compilation behaviour. They also start a new basic block (i.e., they reinitialize the basic block data structures).

The code generator itself consists of instruction selection, instruction scheduling, register allocation, and a final output pass, and is described in Section 4.4. Currently, all stack items reside in a classical memory stack data structure at the basic block boundaries. So, the code generator also generates code to load the needed stack items at the basic block start, to store the produced stack items afterwards, and (if necessary) to update the stack pointers.

Concerning memory allocation: While a basic block is being processed, data is only allocated and never freed. After the code for a basic block is generated, the memory used by its data structures is reclaimed.

4.2 Dealing with unusual compilation behaviour

A traditional threaded-code-based Forth system compiles a normal word[4] is compiled by putting the execution token of the word into the dictionary with `,`. In a native code compiler, the compilation of such words often requires unusual behaviour[5]; e.g., in the context of RAFTS, the compilation of some words just has an effect on the compile-time data stack, for other words the compiler builds unary DFG nodes, for others binary DFG nodes etc. Several approaches have been used for solving this problem:

Intelligent `COMPILE,` The Forth system compiles a word by passing the execution token of the word to `COMPILE,`. `COMPILE,` performs the appropriate compilation action for the execution token. With an appropriate representation for the execution token, this approach can be fast and clean; it is used by BigForth[6]. In the prototype RAFTS system, we distinguish between constants, variables, native-code colon definitions and other words in `COMPILE,`.

State-smart words All words that need some unusual compilation behaviour are made immediate and state-smart (i.e., they contain a test of `STATE`). When executed in compile state, they perform their compilation behaviour. This solution has been used by Wonyong Koh [X3J96] and partly in iForth[7]. Unfortunately it is not standard: e.g., the execution token represents a state-smart semantics, not the interpretation semantics [X3J96].

Wordlists There is a separate wordlist that is searched by the text interpreter in compile state. Words with unusual compilation behaviour have namesakes in this wordlist, that are immediate and perform the compilation semantics when executed. This approach is used by cmForth and by the current RAFTS prototype. However, extending this approach to deal with multiple programmer-visible wordlists (e.g., the ANS Forth search order wordset) is complicated. The current RAFTS prototype also does not handle the redefinition of a word in the compilation wordlist correctly.

Compilation tokens For every name, the Forth system provides an execution token and a compilation token. For interpretation, the text interpreter executes the execution token, for compilation the compilation token. The compilation token for words with unusual compilation behaviour represents that behaviour. This approach is used by DynOOF[8] and partly by iForth.

Generally speaking, we start out with a text string representing the name of a word and end up with performing either the interpretation behaviour or the compilation behaviour. The four approaches differ in the point in time when the information present at the start is reduced so much that only the compilation behaviour can be performed: in the *wordlist* approach this happens during the dictionary search; in the *compilation token* approach it happens during the conversion from the NFA (name field address, or, in general, the name identifier) to the appropriate token; in the *intelligent `COMPILE,`* approach it happens later, during `COMPILE,`; and in the *state-smart* approach it happens during execution.

The problem of the *state-smart* approach is that it tries to resolve interpretation/compilation when the information necessary for this resolution is no longer available. On the other hand, the early resolution in the *wordlist* and *compilation token* approaches can lead to inefficiency: The code produced by `' + compile,` will be less efficient than the code produced by the regular compilation of `+`.

[4] In standard terminology, a word with default compilation semantics.

[5] Note that this unusual behaviour must implement the default compilation semantics (as defined in the standard). E.g., the compilation semantics of `+` is to append the execution semantics of `+` to the current definition. A standard system must implement these compilation semantics. The term *unusual compilation behaviour* here just means that these compilation semantics are implemented in an unusual and nonuniform way (the usual, uniform way is to `,` the execution token).

[6] Email communication with Bernd Paysan, 1996.

[7] Email communication with Marcel Hendrix, 1996.

[8] comp.lang.forth posting by Andras Zsoter, 1996.

Therefore, the optimum approach is the *intelligent COMPILE,*.

As mentioned above, RAFTS currently employs a combination of the *wordlist* and the *intelligent COMPILE,* approach.

4.3 Data flow graph details

There are a few complications:

- Some words (e.g., `!`) push nothing on the stack, but they cannot be optimized away, because they have side effects in memory. These nodes cannot be found by performing a bottom-up graph-walk (bottom-up with respect to Fig. 1). The compiler keeps these nodes in a separate list.
- Variables and Constants are simply executed during compile time. The address or the constant value is inserted into the DAG as a literal. Literals are represented by leaf nodes.
- The compiler must not use the regular data, return or floating-point stack for the symbolic execution, because the programmer may use them at compile time; the Forth standard specifies that words with default compilation semantics (like the words constituting basic blocks) have the compile-time stack effect `--` (i.e., no change). So the compiler must maintain separate compile-time stacks and perform the symbolic execution on them. There is a compile-time stack for the data and the return stack (currently our compiler does not deal with floating-point operations).

At the start of a basic block, the compile-time stacks (both, data and return) are initialized with nodes, that represent the stack elements; each stack element is represented by a different node. At present these nodes represent loads of the stack items from the stack in memory, so the code generator can generate the loads automatically. When the basic block terminates, it is possible to check which stack items are fetched, which are changed (and need to be stored), and how much the stack pointers have changed.

Before the code generator is invoked, stores of the changed stack items into the memory stacks and nodes representing the stack pointer updates are inserted into the data flow graph.

4.4 Code Generator

The three steps to generate the machine code are:

Instruction selection: We use tree parsing for instruction selection. To generate the tree parsing automaton, we use a version of the Burg tree parser generator [FHP91] (modified to generate Forth code instead of C code). The tree parser works in two passes over the DFG: the first pass computes a state for each node and is performed during DFG building; the second pass actually builds the instruction DAG.

Instruction scheduling: List scheduling [LDSM80] is used to transform the instruction DAG into a linear list of instructions.

Register allocation: After scheduling, register allocation becomes trivial: the compiler simply keeps track of unused registers, allocates one when needed, and returns it to the free pool when the value it contains is no longer needed.

4.5 Calls and colon definitions

The native code for a colon definition starts at the body (aka parameter field address, PFA) of the word.

```
#machine code for a native code call
... machine code ...
jal <PFA of the word>   #call word
nop                     #delay slot
... machine code ...
```

Fig. 3. Call of a native code definition

A native code definition calls another native code definition with a `jal` instruction to the PFA of the called word (see Fig. 3). This instruction saves the return address in the register `%ra` (`$31`). A native code word must save its return address, if it calls other words. So, the native code of a definition is wrapped into two small code sequences: the init sequence saves the return address on the return stack. The exit code sequence restores the return address from the return stack and jumps to the return address (see Fig. 4).

`RECURSE` is just a call to the current word.

4.6 Control structures

The control structures are based on a small set of control flow words: `IF`, `AHEAD`, `THEN`, `BEGIN`, `UNTIL`, `AGAIN`, and `CS-ROLL`. ANS Forth allows the creation of arbitrary control structures using these words [Bad90, ANS93]. In the RAFTS prototype, all other control structures (including the counted loops (e.g., `DO..LOOP`) are created from these basic control structure words.

All basic control flow words except `CS-ROLL` end the current basic block. `IF`, `AHEAD`, `UNTIL` and

```
: sum ( n1 -- n1 )
  dup 1+ * 2/ ;

PFA:
#init code sequence
sw %ra, -4(%rp)     #save return address
addui %rp, %rp, -4 #update rp

#machine code of the word sum
lw $2, 0(%sp)       #load stack item
nop                 #load delay slot
addu $3, $2, 1      #1+
multu $2, $3        #*
mflo $4             #get multu result
sra $2, $4, 1       #2/
sw $2, 0(%sp)       #store stack items

#exit code sequence
lw %ra, 0(%rp)      #restore return address
addui %rp, %rp, 4  #update rp
jr %ra              #return
nop                 #branch delay slot
```

Fig. 4. A colon definition in native code

AGAIN represent branches and insert a branch node into the DFG before ending the basic block. The addresses for resolving the branches are passed through the control flow stack, as in other Forth systems.

4.7 Integration with threaded code

Threaded code uses two special registers. `%ip` (`$20`) is the Forth instruction pointer (should not be confused with the instruction pointer of the processor). `%cfa` (`$16`) holds the code field address (CFA).

The native code compiler is integrated thoroughly with the threaded-code-based Gforth system [Ert93]: it is possible to call both native code and threaded code words from both native and threaded code.

This makes the native code compiler much simpler: We don't have to explicitly implement I/O words, floating-point words, or potentially complex control structures like `EXECUTE` and `CATCH..THROW`, but we can still use them.

Calling native code from threaded code. Like the `docol` definition handler in Gforth, the native code word has its own definition handler: `docode`. The inner interpreter of Gforth can execute a native code definition through this handler.

`docode` saves the Forth instruction pointer at the return stack. The address of the native code definition is calculated from the code field address. Then it calls the native code definition with a `jalr`. When the native code definition returns, the handler restores the instruction pointer from the return stack. Then it performs a NEXT (see [Ert93, Section 3]).

Fig. 5 shows `docode` for the direct threaded version of Gforth.

```
#machine code before the call
addui %cfa, %cfa, 8 #compute addr of NC
sw %ip, -4(%rp)     #save ip
jalr %cfa, %ra      #call the NC word
addui %rp, %rp, -4  #delay slot: update rp

#machine code after the call
lw %ip, 0(%rp)      #restore ip
addiu %rp, %rp, 4   #update rp
lw, %cfa, 0(%ip)    #NEXT, direct threaded
addiu %ip, %ip, 4
jr %cfa
nop
```

Fig. 5. `docode` (direct threaded)

Calling threaded code from native code. A native code word calls a threaded code word in the following way: The execution token (CFA) of the word to be called is loaded into the `%cfa` register. The `%ip` register is set to point to a cell that contains the address of the native code that follows the call (for direct threading; add one indirection level for indirect threading). Then the word is called by jumping to the code field of the threaded code word[9] (see Fig. 6). Upon returning, it will execute the word pointed to by `%ip`, which is the rest of our native code definition.

```
#machine code for call to threaded code
    ... machine code ...
    li %cfa, <CFA of the word>
    li %ip, <L0>
    jr %cfa #execute threaded code word
    nop
L0: <L1>
L1: ... machine code ...
```

Fig. 6. Call of a direct threaded code word

[9] In contrast, a native code word calls a native code word by performing a machine-code call (for the MIPS architecture, a `jal`) to PFA of the native code word (see Fig. 3).

4.8 CREATE...DOES>

The DOES>-part is compiled as a separate, anonymous colon definition.

A word defined by a defining word with DOES> gets a special definition handler: docodedoes. When threaded code executes such a word, the processor jumps to docodedoes, which first pushes the PFA of the defined word on the data stack, and then calls the DOES>-part in the same way docode calls a native code definition.

A native code definition invokes such a word by pushing the PFA of the word (like a literal) and then calling the DOES>-part.

4.9 Architecture-specific problems

Currently a nop instruction is inserted after all loads and branches, except for the loads of the stack items, which have only a nop after the last load. The MIPS architecture requires that the load delay slot be filled with an independent instruction (e.g., a nop) and it also specifies a branch delay slot.

The MIPS architecture also requires an instruction-cache flush between storing a piece of native code and executing it. The RAFTS prototype performs an instruction-cache flush when it completes a definition.

5 Statistics

This section shows some statistics for the code generated by the current compiler. The program compiled to gather these results is the compiler itself. The statistics do not reflect the execution frequency of the code. Figure 7 shows counts of various word classes.

5.1 Basic block length

We consider the loads (and their nops) and stores of stack items, and stack pointer updates as *overhead*, because we intend to eliminate them in the final RAFTS compiler. We call the other instructions *useful instructions*.

Only few basic blocks have more than two useful instructions (see Fig. 8 and Fig. 9). On average, a basic block has 2 useful and 3.33 overhead instructions.

5.2 Loads, stores, and stack pointer updates

There is less than one stack pointer update on the average at a basic block (see Fig. 8 and Fig. 10), because there is at most one update per stack pointer and basic block, and often one or both stack pointers do not change.

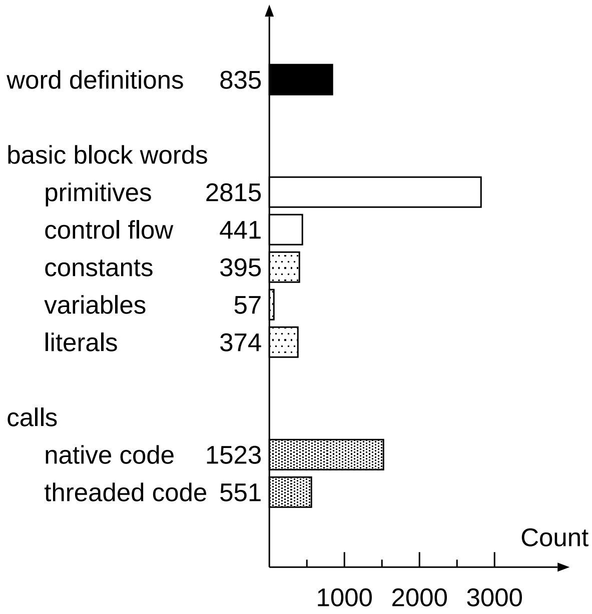

Fig. 7. Number of definitions and invocation frequency of word classes

useful instructions	basic blocks	stack ptr updates	stack loads	stack stores	total length
0	512	0.96	1.32	1.12	3.38
1	348	0.82	1.82	1.37	5.01
2	696	0.73	0.77	1.11	4.61
3	101	0.89	2.01	1.83	7.73
4	123	0.68	1.57	1.28	7.53
5	56	0.75	2.07	1.29	9.11
6	65	0.28	2.08	0.72	9.08
7	31	0.61	1.84	1.16	10.61
8	22	0.55	1.64	1.00	11.18
9	5	0.80	3.20	1.20	14.20
10	4	1.00	2.00	3.00	16.00
11	9	0.67	2.78	2.00	16.44
15	2	0.00	5.00	2.00	22.00
16	1	1.00	2.00	0.00	19.00
17	2	1.50	3.00	2.00	23.50
20	1	0.00	3.00	1.00	24.00
21	1	1.00	2.00	1.00	25.00
51	1	0.00	2.00	0.00	53.00
total: 3954	1980	1585	2657	2384	10560
avg.: 2.00	—	0.80	1.34	1.20	5.33

Fig. 8. Frequency of the different basic block lengths and average overhead instructions

Some basic blocks have no useful instructions. They consist only of overhead instructions (e.g., a basic block consisting of only a swap causes two loads and two stores, with no useful instruction).

The overhead is quite independent of the basic block length. Therefore two strategies can be used to reduce the overhead:

make fewer, bigger basic blocks: Inlining small words eliminates some loads and stores

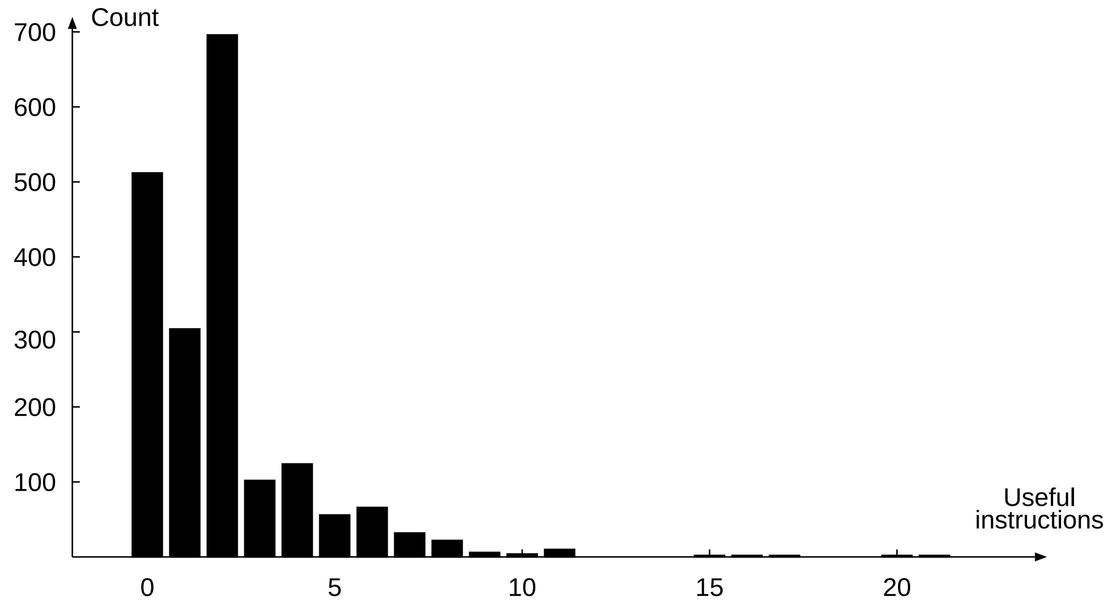

Fig. 9. Frequency of the different basic block lengths

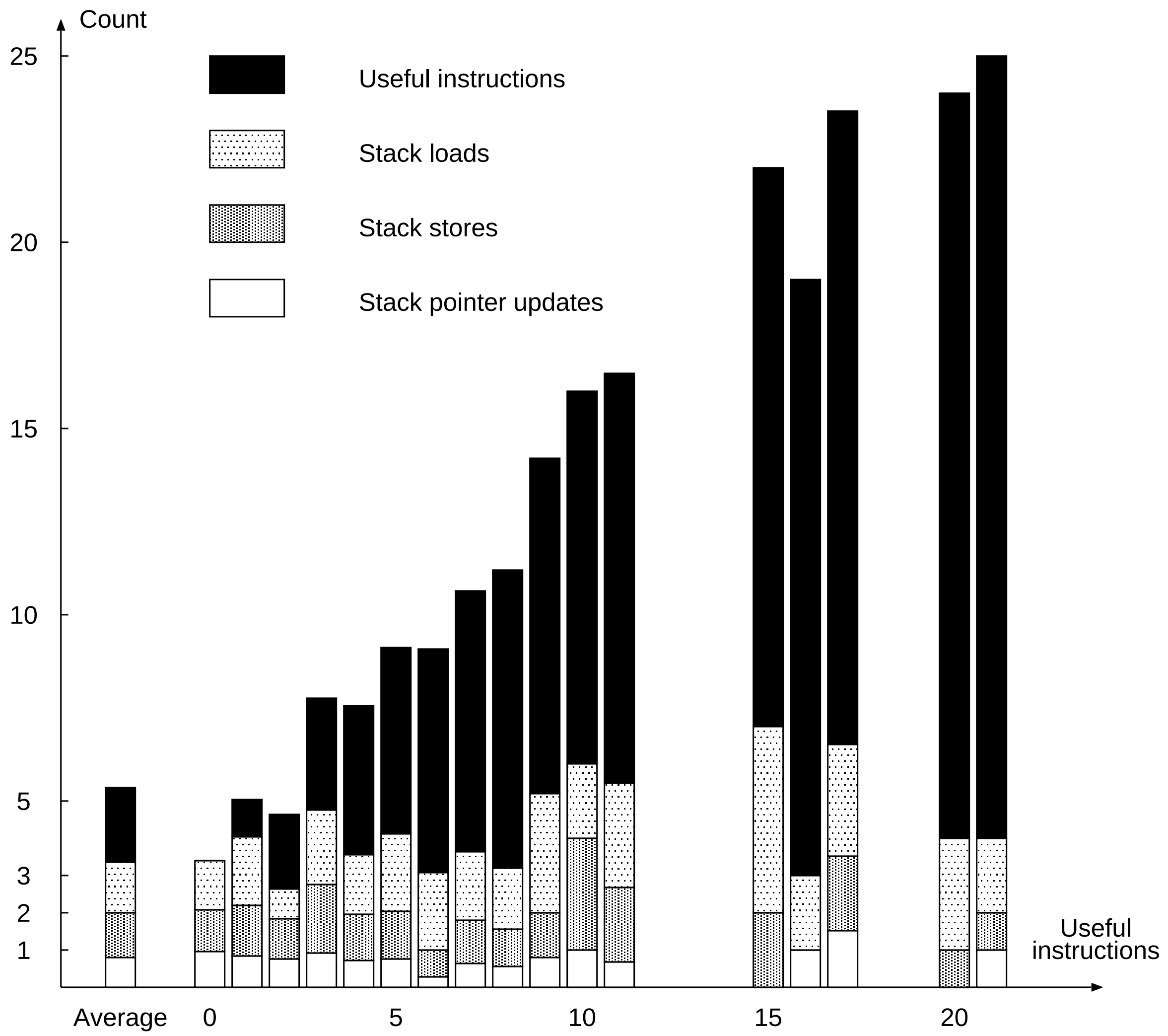

Fig. 10. Average overhead instructions of the different basic block lengths

at block boundaries. It also reduces the stack pointer updates.

reduce the overhead per basic block: If we hold the top of stack element (TOS) in a register across block boundaries, we can eliminate some loads and stores at the block boundaries. The stack pointer updates are still necessary. Interprocedural register allocation can eliminate nearly all loads and stores of stack items, and nearly all stack pointer updates.

6 Benchmark results

The times were taken on a DecStation 5000/150 (50/100 MHz R4000) under Ultrix.

6.1 Benchmarks

The benchmarks we used were the ubiquitous Sieve (counting the primes < 16384 a thousand times); bubble-sorting (6000 integers) and matrix multiplication (200×200 matrices) come from the Stanford integer benchmarks (originally in Pascal, but available in C[10]) and have been translated into Forth by Martin Fraeman and included in the TILE Forth package. These three benchmarks share one disadvantage: they have an unusually low amount of calls. To benchmark calling performance, we compute the 34^{th} Fibonacci number using a recursive algorithm with exponential run-time complexity.

relative times	forth2c opt.	RAFTS	Gforth	forth2c opt. times
sieve	1.00	2.19	6.59	2.7s
bubble	1.00	2.29	6.71	3.4s
matmul	1.00	1.18	5.57	2.8s
fib	1.00	1.39	4.33	4.6s

Fig. 11. Benchmarks: The RAFTS compiled benchmarks in comparison with some other systems on a DecStation 5000/150

The run-time of the benchmarks are compared with the direct-threaded Gforth programs, and the C-code generated by forth2c [EM95] (compiled with `gcc-2.4.5 -O3 -fomit-frame-pointer`) (see Fig. 11).

The slowdown over forth2c is 1.18–2.29, a little less than for the native code compilers measured in [EM95] (on an Intel 486), but there is still a long way to go. The speedup over Gforth (direct threading, without TOS in register) is 2.93–4.72.

[10] The C version and Martin Fraeman's original translations to Forth can be found at http://www.complang.tuwien.ac.at/forth/stanford-benchmarks.tar.gz (if you only have ftp: ftp://ftp.complang.tuwien.ac.at/pub/forth/stanford-benchmarks.tar.gz)

6.2 A large Benchmark: Compilation

As a larger benchmark, we compiled the RAFTS prototype with itself (see Fig. 12). For this benchmark, the RAFTS compiler can run in threaded code or in RAFTS-compiled native code. The speedup of native code over threaded code is a factor of 1.45; We attribute this disappointing speedup to the high number of calls to threaded code words.

relative time		producing: threaded code	producing: native code
running	threaded code	1.00	9.44
	native code	—	6.52

Fig. 12. Relative compile times of different compilers executed with different execution techniques (compiling the RAFTS prototype)

We can also compare the compile times of Gforth's compiler and our RAFTS prototype: the RAFTS prototype (running in native code) compiles 6.52 times slower; compilation speed has not been a primary goal (yet).

7 Conclusion

The current RAFTS system produces code for the MIPS architecture. It runs on top of the Gforth threaded code system and is well integrated with it. Currently there is no register allocation across basic blocks. This results in a great amount of overhead for loading and storing stack items at basic block boundaries and for stack pointer updates: on average, a basic block consists of 2 useful and 3.33 overhead instructions. Consequently, the performance is far from optimal: the code produced by RAFTS is 1.18-2.29 times slower than the code produced by forth2c/`gcc`. This is hardly better than conventional native code compilation techniques. Therefore, implementing RAFTS without good register allocation and/or inlining is not worth the trouble.

We are working on adding both optimizations to RAFTS, on targeting it to the 386 and Alpha architectures, and on achieving competitive compilation speed.

Acknowledgements: Marcel Hendrix and the referees provided valuable comments on draft versions of this paper. Tom Almy, Mike Haas, Marcel Hendrix, Mike Hore and Bernd Paysan discussed their native code compilers with me.

References

[Alm86] Thomas Almy. Compiling Forth for performance. *Journal of Forth Application and Research*, 4(3):379–388, 1986.

[ANS93] *Draft proposed American National Standard — Forth (X3J14 dpANS6)*, 1993.

[AS92] Kristy Andrews and Duane Sand. Migrating a CISC computer family onto RISC via object code translation. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 213–222, 1992.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bad90] Wil Baden. Virtual rheology. In *FORML'90 Proceedings*, 1990.

[CU89] Craig Chambers and David Ungar. Custumization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–160, 1989.

[EM95] M. Anton Ertl and Martin Maierhofer. Translating Forth to efficient C. In *EuroForth '95 Conference Proceedings*, Schloß Dagstuhl, Germany, 1995.

[Ert92] M. Anton Ertl. A new approach to Forth native code generation. In *EuroForth '92*, pages 73–78, Southampton, England, 1992. MicroProcessor Engineering.

[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*. Dissertation, Technische Universität Wien, Austria, 1996.

[FHP91] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG — *Fast Optimal Instruction Selection and Tree Parsing*, 1991. Available via anonymous `ftp` from `kaese.cs.wisc.edu`, file `pub/burg.shar.Z`.

[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.

[KB92] Andreas Krall and Thomas Berger. Fast Prolog with a VAM_{1p} based Prolog compiler. In *Programming Language Implementation and Logic Programming (PLILP '92)*, pages 245–259. Springer LNCS 631, 1992.

[Kna93] Peter J. Knaggs. *Practical and Theoretical Aspects of Forth Software Development*. PhD thesis, School of Computing and Mathematics, University of Teesside, Middlesbrough, Cleveland, UK, March 1993.

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

[Koo92] Philip J. Koopman, Jr. A preliminary exploration of optimized stack code generation. In *1992 Rochester Forth Conference*, 1992.

[LDSM80] David Landskov, Scott Davidson, Bruce Shriver, and Pattrick W. Mallet. Local microcode compaction techniques. *ACM Computing Surveys*, 12(3):261–294, September 1980.

[Pay91] Bernd Paysan. Ein optimierender Forth-Compiler. *Vierte Dimension*, 7(3):22–25, September 1991.

[Pir95] Christian Pirker. Übersetzung von Forth in Maschinensprache. Diplomarbeit, Technische Universität Wien, 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.

[Sto88] Bill Stoddart. Specification & optimisation. In *euroFORML'88 Conference Proceedings*, pages 147–165, MPE Ltd, 133 Hill Lane, Southampton SO1 5AF, UK., September 1988. Forth Interest Group.

[Tin81] C. H. Ting. *Systems Guide to fig-Forth*. Offete Enterprises, Inc., San Mateo, CA 94402, 1981.

[TvSKS83] Andrew S. Tanenbaum, Hans van Staveren, E. G. Keizer, and Johan W. Stevenson. A practical tool kit for making portable compilers. *Communications of the ACM*, 26(9):654–660, September 1983.

[Ung87] David Ungar. *The Design and Evaluation of a High-Performance Smalltalk System*. MIT Press, 1987.

[X3J96] TC X3J14. Clarifying the distinction between "immediacy" and "special compilation semantics". RFI response X3J14/Q0007R, ANSI TC X3J14, 1996.