A Preliminary Exploration of Optimized Stack Code Generation

Philip Koopman, Jr.

United Technologies Research Center
411 Silver Lane M/S 48
East Hartford, CT 06108
koopman@utrcgw.utc.com

ABSTRACT

This paper presents an experimental code generator that performs intra-block stack scheduling for a stack-based execution model. For small test programs, 91% to 100% of redundant local variable accesses were eliminated using this compiler. Compiled intra-block stack scheduling and hand-performed global stack scheduling show that significant opportunities exist to keep temporary variable values on the expression evaluation stack when compiling conventional languages.

INTRODUCTION

Efficient compilation of conventional languages to stack-based execution models has received relatively little attention. Previous work has focused on using the stack efficiently for expression evaluation (*e.g.*, Bruno & Lassagne (1975), Couch & Hamm (1977), Miller (1987)) and performing peephole optimizations (*e.g.,* Hayes (1986), Hand (1989)). There seems to have been little work on optimizing stack usage at the basic block level (code sequences containing no branches nor labels) or the global level (whole-procedure).

One significant advantage of register-based computation models is that register scheduling algorithms can place temporary variables in registers instead of memory-based variables to improve code efficiency. A common criticisms of stack-based execution is that variables can't be kept on the stack without a lot of wasted stack manipulation. There is data to back this up: Koopman (1989) gives dynamic instruction frequencies for Forth programs indicating approximately 1 operation in 4 is a DUP, SWAP, or other stack manipulation.

The question is, can stack-based computational models use stacks for temporary values as effectively as register-based computational models? While there are not yet any definitive answers, this paper seeks to establish a foundation for performing experiments on stack-based code generation. It does so by using an experimental stack-scheduling compiler and hand-coding techniques to optimize stack usage on several small C programs at both the basic-block level and the global level.

METHODOLOGY

I have developed an experimental stack-based code generator that works as a postprocessor for the GNU C compiler (Stallman 1988). In order to compile a program, GNU C is run on the source code with optimization enabled. A dump of the intermediate code data structures (in LISP-like notation) is generated just before the register assignment pass of GNU C. The resulting dump file is used as input for the stack-based code generator. The output of the code generator is Forth source code augmented with stack comments on every line.

The results reported here are for compiling seven integer benchmark programs from the Stanford Benchmark Suite (Hennessy & Nye). Although significantly larger programs must be used to make conclusive statements about performance, these programs are sufficiently complex and varied at the basic block level to form a starting point for experimentation. The goal of my experiment was to explore the area with limited resources rather than give definitive results.

GENERAL COMPILATION

The stack-based code generator works in six steps:

1) Raw input processing:
The LISP-notation input file is parsed to generate a list of stack-based instructions using numbered local variables (the temporary register numbers from GNU C are the local variable numbers). Register-to-register operations in the input are transformed into loads of operands from local variables to the stack, an operation, and a store of the result back into a local variable. For example:

```
(insn 42 41 43
   (set (reg:SI 77)
        (plus:SI (reg:SI 75)
                 (reg:SI 73))) -1 (nil)
   (expr_list:REG_DEAD (reg:SI 73)
   (nil)))
```

is translated into:

```
(          --)  75 LOCAL@
(       75 --)  73 LOCAL@
(    75 73 --)  +
(       77 --)  77 LOCAL!
```

A running stack picture is maintained as the code is translated. Each instruction in the stack-code list gets a copy of the stack picture as it is before the instruction's stack effect. The stack is always empty at the end of each GNU C LISP expression.

The example expression above declares the value in register 73 to be "DEAD". This means the value is not needed in subsequent expressions. The "DEAD" attribute is captured as a pseudo-operation in the stack-code list for later use to eliminate unnecessary stores to local variables whose values are not reused.

2) Code clean-up:
This is a set of related steps that modifies code details to finish converting from a register-based paradigm to stack operation. Modifications include holding subroutine inputs on the stack instead of in registers, modifying condition code phrases to use stack-based comparisons, and ensuring subroutine return values are placed on the stack.

3) Peephole optimization:
An initial pass of a peephole optimizer is used to make the code more consistent for processing in later phases. Usually very few changes take place here.

4) Stack scheduling:
This will be presented in more detail in subsequent sections. The purpose of stack scheduling is to carry values on the stack instead of using local variables.

5) Peephole optimization:
The peephole optimizer is used again to clean up the code. It is typically much more effective on this pass than on the first pass, and is used to simplify complex stack manipulations that may result from code scheduling. For example:

DUP SWAP becomes DUP
SWAP TUCK becomes OVER

Also, stores to dead local variables are eliminated in this pass by using transformations such as:

DUP (dead)LOCAL! becomes NOP

6) Code generation:
In this phase mapping from stack-code to target machine instructions is performed. Composite instructions are also generated here (see Appendix for examples). For example:

OVER + becomes OVER_+
TUCK ! becomes TUCK_!

INTRA-BLOCK STACK SCHEDULING

The stack scheduling step described in the previous section can be performed at two levels: intra-block scheduling, which will be discussed in this section; and global scheduling, which will be discussed in the next section.

Intra-block stack scheduling attempts to remove local variable fetches and stores by maintaining copies of the local variables on the stack. It designates where variables go on the stack for each instruction. The terminology is deliberately similar to "register scheduling", in which variables are assigned to registers in conventional compilers (*e.g.,* Hennessy & Patterson (1990), pp. 113-114).

Generalized stack scheduling is a bit more difficult than register scheduling because stack depth must be uniform and consistent when control passes through branch targets from differing branch sources. For example, in an IF...ELSE...THEN construct, the IF and ELSE clauses must both leave the same temporary variables in the same locations on the stack when control transfers beyond the THEN. The simplest way to ensure that this happens is to adopt a policy of always leaving the stack empty when taking a branch. An obvious way to implement this policy is to employ stack scheduling only within basic blocks (*intra-block* stack scheduling).

The Intra-Block Scheduling Algorithm

I experimented with several intra-block scheduling algorithms while performing compiler research for the Harris RTX 2000 stack CPU. All of the methods I used then were based on ad-hoc sets of transformations that seemed to work, but lacked a unifying approach. However, recent exploration of the problem from a fresh start has yielded a simple heuristic approach that seems to give excellent results.

The algorithm for intra-block stack scheduling has two parts. The first part is ranking opportunities for eliminating local variable fetches and stores. The second part is attempting to actually eliminate the local variable operations in rank order.

An opportunity for stack scheduling exists whenever a local variable value is used more than once within a basic block. We will use code generated from the following code sequence as an example of operation:

```
b = a + c;
a = b + 5;
c = b + a;
```

This code sequence could be compiled into the following stack code:

```
(       --)  75 LOCAL@    \ a @
(    75 --)  77 LOCAL@    \ c @
( 75 77 --)  +
```

```
(    76 --)  76 LOCAL!   \ b !
(       --)  76 LOCAL@   \ b @
(    76 --)  5           \ literal 5
( 76 88 --)  +
(    75 --)  75 LOCAL!   \ a !
(       --)  76 LOCAL@ (DEAD) \ b @
(    76 --)  75 LOCAL@   \ a @
( 76 75 --)  +
(    77 --)  77 LOCAL!   \ c !
```

where the variable **a** is assigned to local variable 75, **b** is assigned to 76, and **c** is assigned to 77. Note that because of data dependencies, rearrangement of the assignments can't remove the need to keep a temporary copy of the value of **a** in **b** for computation of the new value of **c**. Nevertheless, there is an opportunity to perform stack scheduling on local variables 75 and 76 because they are each used multiple times as operands in the computation.

The first step in the stack scheduling process is to measure the distance between a local variable fetch operation and the nearest preceding occurrence of that variable on the stack. The idea is to schedule the closest "use/reuse pairs" first, because they have a high likelihood of success. The distance measures for our example are as shown:

```
(       --)  75 LOCAL@
(    75 --)  77 LOCAL@
( 75 77 --)  +
(    76 --)  76 LOCAL!  ──────────┐
(       --)  76 LOCAL@  ──────────┘ 1
(    76 --)  5
( 76 88 --)  +  ──────────────┐ 2
(    75 --)  75 LOCAL!  ──────┤
(       --)  76 LOCAL@ (DEAD) ─┘  2
(    76 --)  75 LOCAL@  ──────────┘
( 76 75 --)  +
(    77 --)  77 LOCAL!
```

Variable 76 is reused immediately after it is stored, with a distance measure of 1. Variable 76 is reused again with a single intervening instruction, giving a distance measure of 2. Variable 75 is reused with a distance measure of 2 as well.

After distances are measured, candidates for stack scheduling (use/reuse pairs) are ranked in order of ascending distance. Thus, the algorithm considers nearest pairs before more distant pairs. I chose this ranking method so that nested uses of values on the stack would be scheduled optimally (from the inner usages out).

For each pair of use/reuse stack operations, the following procedure is applied. The register of interest must be able to be copied to the bottom of the stack using: DUP, TUCK, UNDER, or TUCK_2 (see Appendix), otherwise the scheduler moves on to the next use/reuse pair. Also, the stack depth at the reuse instruction must also be 2 or less (so that the register of interest can be brought from its new position on the bottom of the stack to the top with a SWAP or ROT if necessary). If both of these conditions hold, a DUP, TUCK, UNDER, or TUCK_2 instruction is inserted before the "use" instruction to copy the value to the bottom of the stack, and the LOCAL@ instruction at the "reuse" point is replaced with a NOP, SWAP, or ROT as appropriate.

The idea behind the stack scheduling algorithm is to copy a value to the bottom of the stack at the use point, update the stack pictures of the instructions between the use and reuse points, and move the value back to the top of the stack at the reuse point. As successive use/reuse pairs with longer distance measures are scheduled, their values tend to go *under* the existing scheduled values on the stack.

Access is limited to only the top 3 stack elements at use/reuse points. My previous experience indicates arbitrary-depth PICKs generate poor code because of subsequent DROP operations required to eliminate dead values. Also, many stack CPUs pay performance penalties for deep stack accesses. Note that the stack itself may get deeper than three elements (the small programs I compiled used up to 8 stack elements); it is just access to the stack that is limited to the top 3 values.

In our example, the distance-1 use/reuse pair is a reuse of the value of variable 76. This is transformed into:

```
(       --)  75 LOCAL@
(    75 --)  77 LOCAL@
( 75 77 --)  +
(    76 --)  DUP         \ copy of 76
( 76 76 --)  76 LOCAL!
(    76 --)  NOP         \ 76 LOCAL@
(    76 --)  5
( 76 88 --)  +
(    75 --)  75 LOCAL!
(       --)  76 LOCAL@ (DEAD)
(    76 --)  75 LOCAL@
( 76 75 --)  +
(    77 --)  77 LOCAL!
```

Although distances between uses/reuses may be changed by the transformations, the distance metric need not be recomputed. Next is the distance-2 opportunity for the reuse of variable 76. This use/reuse pair is selected instead of the variable 75 pair with distance measure 2 because it comes sooner in the instruction

list; in practice which of an overlapping set of equal distance measures is picked doesn't seem to make much difference. After transformation, the example becomes:

```
(           --)  75 LOCAL@
(        75 --)  77 LOCAL@
(     75 77 --)  +
(        76 --)  DUP
(     76 76 --)  76 LOCAL! (DEAD)
(        76 --)  NOP
(        76 --)  5
(     76 88 --)  UNDER       \ copy of 76
( 76 76 88 --)  +
(     76 75 --)  75 LOCAL!
(        76 --)  NOP         \ 76 LOCAL@
(        76 --)  75 LOCAL@
(     76 75 --)  +
(        77 --)  77 LOCAL!
```

Note that since the fetch of local variable 76 was eliminated, the DEAD annotation migrates to the previous variable 76 reference. This shows that the store of local variable 76 is in fact unnecessary (because variable **b** is only a temporary holding variable).

Finally, the distance-2 opportunity for variable 75 is exploited giving the output of the code scheduler:

```
(           --)  75 LOCAL@
(        75 --)  77 LOCAL@
(     75 77 --)  +
(        76 --)  DUP
(     76 76 --)  76 LOCAL! (DEAD)
(        76 --)  NOP
(        76 --)  5
(     76 88 --)  UNDER
( 76 76 88 --)  +
(     76 75 --)  TUCK        \ copy of 75
( 75 76 75 --)  75 LOCAL!
(     75 76 --)  NOP
(     75 76 --)  SWAP        \ 75 LOCAL@
(     76 75 --)  +
(        77 --)  77 LOCAL!
```

There are of course many inefficiencies in stack manipulation in this code. The peephole optimizer uses a few dozen simple rules to eliminate inefficient code sequences. I found it was easier just to enumerate the situations that arose in practice rather than "teach" efficient stack usage to the compiler. The following peephole rules are used on the above code:

DUP (dead)LOCAL! becomes NOP
SWAP + becomes +
TUCK LOCAL! + becomes DUP LOCAL! +
NOP is deleted

giving the result:

```
(           --)  75 LOCAL@
(        75 --)  77 LOCAL@
(     75 77 --)  +
(        76 --)  5
(     76 88 --)  UNDER
( 76 76 88 --)  +
(     76 75 --)  DUP
( 76 75 75 --)  75 LOCAL!
(     76 75 --)  +
(        77 --)  77 LOCAL!
```

To put things in a more Forth-like representation, this code could be written as:

```
a  c   +        (-- b )
5  UNDER        (-- b b 5 )
+  DUP   -> a (-- b a )
+  -> c        (-- )
```

The resultant code now has 4 local variable references instead of the original 8 and is two instructions shorter. Further reduction is not possible if we assume that the example is a complete basic code block, because the stack must be empty at the start and end of the block.

Note that there are only two stack operations in this code sequence, and both of them (the UNDER and the DUP) could be combined with other operations to form new primitives: UNDER_+ and DUP_LOCAL! . Both these primitives are typically found on stack-based CPUs. In general this code generation process results in very little wasted manipulating of the stack.

Intra-Block Scheduling Results

It is well known that basic blocks in C programs tend to be rather short. This limits the effectiveness of intra-block stack scheduling. However, within the single-block constraints my stack scheduling algorithm tends to produce fairly good code. The code is not necessarily the same as that which would be produced by a human programmer, but is usually very close in efficiency (measured by use of the stack instead of local variables). One reason for the difference between generated code and human-written code is that human programmers typically don't attempt to arrange instructions for stack/operation compound instruction creation, whereas my stack scheduling algorithm does.

One way of measuring the success of intra-block scheduling is by counting the number of "reuse" LOCAL@ instructions that were successfully removed by the algorithm as a percentage of all redundant (reuse) LOCAL@ instructions (Figure 1). The results indicate that the algorithm was highly successful,

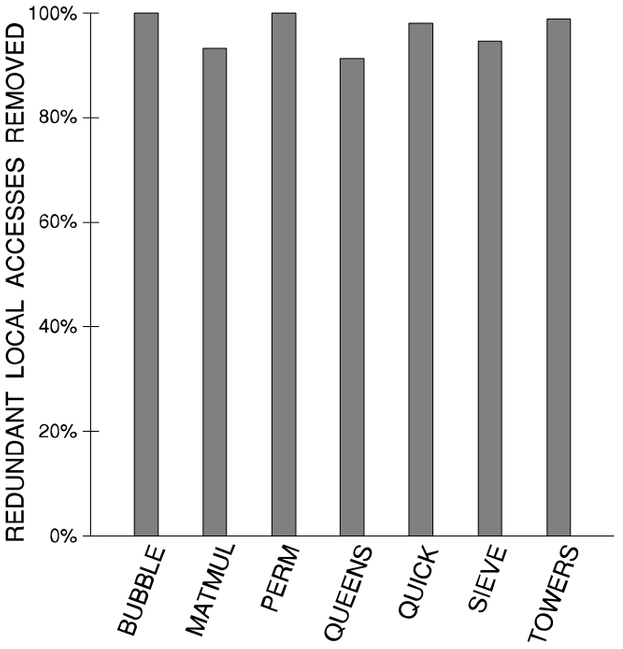

Figure 1. Intra-block stack scheduling removes most redundant accesses to local variables

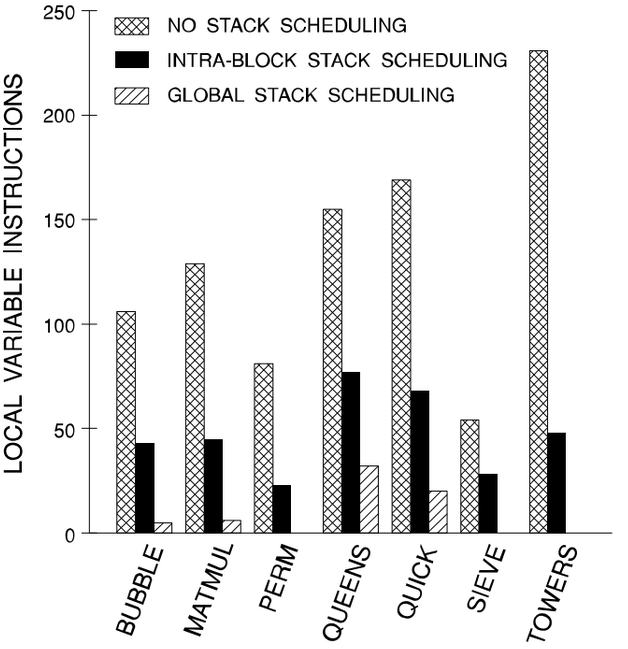

Figure 2. The number of local variable instructions in the compiled code reduces dramatically with stack scheduling.

removing between 91% and 100% of all redundant LOCAL@ instructions.

Surprisingly, deep accesses to the stack were not needed to achieve excellent intra-block scheduling results. ROT was the deepest stack movement operation required, and PICK_2 was the deepest stack element copying operation needed (and neither of these were needed often).

Of course, Figure 1 does not tell the whole story. In determining how well intra-block scheduling does, it is important to look at the total number of local variable accesses. Figure 2 shows the number of local variable loads and stores in each program with no scheduling, intra-block scheduling, and global scheduling (discussed in the next section). While intra-block scheduling removes many of the local variable references, many remain because of variable lifetimes that cross basic block boundaries.

GLOBAL ANALYSIS

Intra-block scheduling has definite limitations. In particular, it is not possible to eliminate local variables whose lifetimes cross basic block boundaries. In order to assess how much these restrictions affected code generation, I hand-generated code using global analysis.

The hand-generated code starting point was the output of the intra-block stack scheduling compiler. I selected variables to keep on the stack using the distance metric across basic block boundaries. In general I was as aggressive as possible in using the stack consistent with keeping accesses to the stack to the top three elements.

Global Scheduling Results

Figure 2 shows that global optimization removed many local variable references beyond those removed by intra-block optimization. For most programs, nearly all references were removed. However, portions of QUEENS and QUICK proved to have too many active values to juggle on the stack. This is not to say that these two algorithms can't be written using entirely stack-resident variables, but rather that the C programs as written in the Stanford Integer Benchmark Suite are difficult to stack-schedule. Although it wasn't entirely successful in eliminating the need for local variables, global scheduling showed significantly improved performance over intra-block scheduling.

I have not created a unified methodology from my experience of hand-performing global stack scheduling -- I just used ad-hoc techniques as necessary. Nonetheless the experience of optimizing several

programs in this manner leads to some observations about how a formal methodology might be created.

Keeping loop indices on the Forth Return Stack is a significant advantage. Also, it is usually fruitful to keep one or sometimes two (but not more) frequently used variables resident on the stack for the entire duration of a loop instead of in local variables. Sometimes placing a dummy value onto the Data Stack in one part of an IF...ELSE...THEN leads to significant simplification of stack manipulation in the other part.

CONCLUSIONS

The algorithm have I developed for intra-block stack scheduling seems to be quite effective, eliminating 91% to 100% of redundant local variable accesses within basic blocks for the small programs studied. Hand-performed global optimization results indicate that significantly better stack scheduling can be done if variables are kept on the stack across basic block boundaries. Global optimization for stack scheduling is still an ad-hoc process. While stack scheduling can eliminate most local variable references, some programs with large numbers of variables are still difficult to stack-schedule.

REFERENCES

Bruno, J. & Lassagne, T. (1975) The generation of optimal code for stack machines. *JACM* , July 1975, **22**(3):382-396

Couch, J. & Hamm, T. (1977) Semantic structures for efficient code generation on a stack machine. *Computer* , May 1977, **10**(5):42-48

Hand, T. (1989) Performance of the Harris RTX-2000 C Compiler. In: *Proc. of the 1989 Rochester Forth Conf., 20-24 June 1989, Univ. of Rochester*, pp. 61-62

Hayes, J. (1986) An interpreter and object code optimizer for a 32 bit Forth chip. In: *1986 FORML Conf. Proc., 28-30 November 1986, Pacific Grove CA*, pp. 211-221

Hennessy, J. & Nye, P., *Stanford Integer Benchmarks*, Stanford University

Hennessy, J. & Patterson, D. (1990) *Computer Architecture: a quantitative approach*, Morgan Kaufmann Publishers, San Mateo, CA

Koopman, P., *Stack Computers*, Ellis Horwood/ Halstead Press, 1989

Miller, D. (1987) Stack machines and compiler design. *Byte* , April 1987, **12**(4):177-185

Stallman, R. (1988) GNU Project C Compiler

APPENDIX: NON-STANDARD PRIMITIVES

The stack scheduler uses primitives not found in many Forth systems. LOCAL@ and LOCAL! are local variable fetch and store operations that take a numeric input (which probably represents an offset into an activation frame) as the "name" of a local variable to be fetched and stored. The following definitions show the behavior of non-standard words discussed. The reader should understand that the intent is for these operations to be provided as quick primitives, not high-level operations.

```
: -ROT  ( a b c -- c a b )
   ROT ROT ;
: OVER_+  ( a b -- a c )
   OVER + ;
: PICK_2  ( a b c -- a b c a )
   >R OVER R> SWAP ;
: TUCK_2  ( a b c -- a a b c )
   PICK_2 -ROT ;
: UNDER  (a b -- a a b )
   >R DUP R> ;
: UNDER_+  ( a b -- a c )
   OVER_+ ;
: DUP_LOCAL!  ( a -- a )
   DUP LOCAL! ;
  ( this assumes that the local
    variable offset is compiled into
    the local variable instruction, not
    left on the stack at run time)
```