Optimizing Stack Code

Martin Maierhofer M. Anton Ertl

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

Abstract

With interest in stack machines recently growing (e.g. the JavaVM architecture used by the Java programming language), support for the efficient execution of Algol-like high level languages on this class of hardware becomes an issue. Local variables accesses in the source language should be translated into stack accesses of the target machine (similar to register allocation on register machines).

In this paper we explore such stack allocation techniques for basic blocks. We evaluate Phil Koopman's "stack scheduling" by adding an instruction scheduler and comparing the result with an optimal stack allocation and instruction scheduling strategy. Stack scheduling in cooperation with a depth first postorder instruction scheduling produces results close to the optimum. The optimizations discussed in this paper are only useful for stack hardware that allows faster access to the stack than to main memory.

1 Introduction

Since Sun introduced the Java programming language and the corresponding virtual machine (JavaVM, a stack architecture [Sun95]), the interest in stack machines once again has increased. After they fell out of favor in the late seventies, stack machines were only used in a relatively narrow, yet important class of applications: consumer products and embedded controllers were their primary domain. Compact code size, low hardware complexity and moderate costs are the main factors responsible for the success of stack machines in that areas ([Koo93]). Support for the efficient execution of Algol-like high level language code was not considered to be of much importance in this domain.

Optimizing compilers for register machines usually employ a technique called "register allocation" to improve code efficiency ([Cha81], [Bri94]). Register allocation maps the variables used in a section of code to the machine's registers in order to reduce access times (in general, access to registers is much faster than access to main memory in register machines). A similar optimization technique can be used for stack based processors.

Most current stack processors make use of a "stack buffer" to cache the topmost stack elements for improved performance ([Koo89]). In analogy to register machines, this makes access to stack elements faster than access to memory. When compiling Algol-like high level languages to stack code, local variables of a function usually are located in the corresponding "stack frame" in main memory (similar to the situation with unoptimized code for conventional register machines).

This creates an opportunity for optimization: instead of loading the contents of a variable from main memory onto the stack each time it is used, the compiler can keep a copy of the variable's value on the stack and reuse this copy in subsequent operations.

An important property of stack machines makes this kind of optimization more difficult than for register machines: stack instructions most often implicitly use the elements at the top of stack as their arguments. If arguments to an instruction already resides in the stack, the stack must be manipulated so that they appear in the order the instruction expects them to be. Stack machines generally possess only a limited set of instructions for stack manipulations, with complex reordering of the stack taking relatively long. Consequently, an optimization technique must deal with that limitations and nevertheless try to minimize the usage of stack manipulations.

This paper evaluates such an optimization technique ("stack scheduling" developed by Phil Koopman) by comparing it with an optimal stack allocator ("optimal DAG scheduling"). The rest of the paper is organized as follows: section two discusses previous publications dealing with the optimization of stack code. Section three gives an overview of the optimization process, while section four contains a more detailed description of the two optimization techniques. Section five presents empirical data and compares the two methods. Section six concludes the paper.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: Forth-Tagung 1997, Ludwigshafen

2 Previous Work

[Bru75] is one of the first publication dealing with the optimization of stack code. The paper presents an efficient algorithm for constructing optimal programs for a limited set of expressions. Similarly, [Pra80] discusses the class of expressions for which optimal stack code can be generated efficiently. The article concentrates on the minimal stack depth required for optimal code.

The table driven peephole optimizer of a portable compiler that uses a stack machine-based intermediate code is discussed in [Tan82]. The paper contains an extensive set of rules for the optimizer and data about their usage in optimizing a large amount of Pascal code. [Mas80] also concentrates on peephole optimization; the authors describe a converter to transform a dialect of Lisp into code for an abstract stack machine.

A C compiler for the NOVIX NC4016/6016 stack processor is discussed in [Mil87]. The compiler makes heavy use of the special features of that processor (e.g. "pseudo registers" supported by the hardware are used as frame pointers). It also supports the opposite of "inlining": semantically equivalent sections of code are compiled into a single subroutine. Because calls can be executed very efficiently by stack processors, this technique reduces code size while retaining performance. Local variables are not cached in the stack but always reloaded from memory; parameters and return values of functions, however, are passed on the stack.

A similar C compiler for the APD MF1600 processor is covered in [Win88]. Unfortunately, the description of the compiler is very superficial and states only that "The translator takes full advantage of a target 'Stack' architecture by [...] allocating storage for all register or auto variables within the Arithmetic Stack." No details are given on the allocation policy or on the actual method to reorder the stack elements appropriately.

Finally, [Koo92] presents "stack scheduling", a technique discussed in depth in section 4.1.

3 Overview

The first author has implemented an optimizer for JavaVM code. The code of each function (or "method" in JavaVM terminology) in a classfile is optimized in the following steps:

1. Because we use local optimization only, the code for a method has to be divided into basic blocks; the basic blocks are the nodes of the data flow graph that is then built for each method. The code of a basic block is guaranteed to be executed from the first to the last instruction in sequential order without change of control flow in between (as an exception, we allow method invocations inside a basic block, because they cannot alter the state of the caller's stack and local variables).

2. A "live variable" analysis is then performed on the data flow graph using an iterative algorithm that solves generic data flow equations ([Aho86]). This step does not actually optimize the code, but it is used to gather information that can be used later on to eliminate stores to "dead" local variables. For each variable, we determine the instruction, which references the variable for the last time when executing the method. The instruction is annotated with information about the variable becoming "dead"; the annotation is then used by the other optimization techniques (notably the peephole optimizer and optimal DAG scheduling).

3. A first pass of the peephole optimizer can improve certain instruction sequences. Basically, this approach follows a simple "search and replace" strategy to eliminate inefficient code. Rules are used to tell the optimizer, which instruction sequence can be replaced by semantically equivalent, but more efficient code. The implementation uses a table driven, flexible peephole optimizer that can be easily extended with new rules.

4. The actual optimization of each basic block is performed by either Koopman's stack scheduling or DAG scheduling as explained in section 4.

5. The peephole optimizer is used to clean up "messy code" that might have been produced by the previous optimization. E.g. code sequences like `swap swap` can be generated by stack scheduling. Instead of complicating the stack scheduling algorithm by trying to "repair" such situations, we let the peephole optimizer remove the unnecessary instructions.

6. Finally, the basic blocks of a method have to be retransformed into a single sequence of stack instructions that can be written to the classfile.

4 Optimization Techniques

Both approaches described in this section are "local" (i.e. each basic block is optimized separately). Because the stack is always empty at the start and end of a basic block,[1] the stack depth will always be consistent. As a consequence, a variable needs to be used more than once in a basic block for these methods to be useful.

[1] At least in theory, because HLL statements are translated into a sequence of stack instructions that starts end ends with the stack being empty. In practice, basic blocks in JavaVM code can start with some elements already on the stack when conditional expressions or exception handling are used.

4.1 Stack Scheduling

4.1.1 The Basic Algorithm

Phil Koopman was the first to present a technique for the kind of optimization discussed in the previous paragraphs—he named it "stack scheduling"[2] ([Koo92]). Basically, stack scheduling substitutes loads of local variables by stack copying and manipulation instructions. Example 1 contains a short fragment of Java code in the first column. The third column shows the corresponding JavaVM code generated by `javac -O` (we assume that `a` is placed at index 1 in the JavaVM local variable table, `b` at index 2 and so forth) and the last one contains equivalent Forth code.

Example 1 A fragment of Java/JavavM code

```
c = a + b;   ( -- )       iload_1    % a @
             ( a -- )     iload_2    % b @
             ( a b -- )   iadd       % +     ┐
             ( c -- )     istore_3   % c !   │ pair
a = b + d;   ( -- )       iload_2    % b @   ┘
             ( b -- )     iload_4    % d @
             ( b d -- )   iadd       % +
             ( a -- )     istore_1   % a !
```

Stack scheduling starts by annotating each instruction of a basic block with information about the stack elements present at run time *before* executing the instruction (the so called "stack picture"). The compiler determines, whether a stack element contains the value of a variable or not. Column two in example 1 shows the annotations that can be inferred easily from the stack code by symbolically executing it.

Next, the algorithm tries to pair each instruction that loads the contents of a local variable onto the stack with another instruction, whose stack picture includes the variable to be loaded. Basically, we walk through the code searching for load instructions. When such an instruction is encountered, the stack pictures of the preceding instructions are searched for an occurrence of the variable referenced by the load instruction. If such an instruction has been found, it is entered into a list of pairs of instructions together with the load instruction, and the algorithm continues to search for further pairs.

In example 1, there are four load instructions. The ones at line one and two load `a` and `b` for the first time respectively, and no "partner" instruction can be found to create a pair. The same holds true for the load instruction at line six, where `d` is loaded for the first and last time in the basic block. When `b` is reloaded at line five, however, the search for a partner instruction is successful: because the stack picture of the `iadd` instruction at line three includes `b`, these two instructions can be paired and inserted into the (hitherto empty) list of pairs.

The next step in the process of stack scheduling consists of sorting the pairs found in the previous step according to the distance between the two instructions. In our example there is not much use in sorting a single pair, but the idea behind sorting at all is that instructions close to one another are more likely to be scheduled successfully.

The final step tries to schedule each pair of the list prepared in the previous steps (i.e. the pairs with a small distance are processed first). A pair of instructions can be scheduled if the following conditions are fulfilled:

- The variable can be copied to the bottom of stack by a stack manipulation in front of the first instruction (the one, whose stack picture includes the variable of interest).

- The copy can be moved from the bottom of stack to the top of stack at the second instruction (the one loading the variable).

If a pair can be scheduled, the appropriate stack manipulation instruction is inserted in front of the first instruction of the pair; then the load instruction can be replaced by another stack manipulation instruction to move the copied stack element to the top of stack (if the stack was empty at the point of loading, there is no need for manipulating the stack and the load can be dropped altogether). After scheduling a pair, the stack picture of all instructions lying in between the first and second instruction of the pair must be updated to include the newly created stack element.

Example 2 contains the code of example 1 after scheduling. Note that `dup_x1` (the same as `tuck` in Forth) has been inserted in front of the first `iadd` instruction (remember that this was the first instruction of the pair we found in example 1). The load instruction has been omitted; because the copy created by `dup_x1` already resides at top of stack at that point, there is no need for a further stack manipulation instruction. There are no further pairs to schedule, so example 2 shows the final result of stack scheduling the code of example 1.

Example 2 Code of example 1 after stack scheduling

```
c = a + b;   ( -- )        iload_1    % a @
             ( a -- )      iload_2    % b @
             ( a b -- )    dup_x1     % tuck
             ( b a b -- )  iadd       % +
             ( b c -- )    istore_3   % c !
a = b + d;   ( b -- )      iload_4    % d @
             ( b d -- )    iadd       % +
             ( a -- )      istore_1   % a !
```

[2] In fact, Koopman does not constrain stack scheduling to be of local scope, but he does not present an algorithm for global stack scheduling either. ("I just used ad-hoc techniques as necessary.")

4.1.2 Extensions

There are two simple extensions to the basic algorithm which are not described in [Koo92]: first, it is not always necessary to put the copy of a variable to the bottom of stack. The rule to be obeyed here is that the copy must not be altered by an instruction executed between the first and second instruction of the pair to be scheduled. Because of the limited set of stack manipulation instructions provided by most stack processors, this extension enables the algorithm to schedule more pairs.

The second extension deals with the search for pairs. While it is desirable that the two instructions of a pair are as close as possible, some situations may require that the partner instruction for a load is not the first one encountered during the search. Consider example 3:

Example 3 A fragment of Java/JavaVM code

```
b = a + 5;    ( -- )       iload_1     % a @
              ( a -- )     iconst_5    % 5
              ( a 5 -- )   iadd        % +
              ( b -- )     istore_2    % b !
c = a;        ( -- )       iload_1     % a @
              ( a -- )     istore_3    % c !
```

The basic algorithm would find only one pair of instructions (namely the instructions in line three and five). Variable `a` cannot be copied to the bottom of stack at the point of occurrence,[3] so this pair is not scheduled. If instead we go on searching for additional partner instructions after finding the first one, another pair is found consisting of the instructions in line two and five. This time, the pair can be scheduled (`dup` is inserted in front of `iconst_5` and the second `iload_1` is eliminated) and the extension to the original algorithm yields optimal code.

When searching for partner instructions of a load instruction, care must be taken not to use an obsolete value of the variable: the search has to be terminated when the examined instruction might alter the variable (this could be an assignment or `inc` instruction).

Situations that can take advantage of the second extensions we described in this section do not occur often. The level of optimization of JavaVM code as described in section 5 did not deteriorate seriously when we switched off the extension. We did not implement the first extension.

4.1.3 Drawbacks

The most prominent problem of stack scheduling is caused by the absence of instruction scheduling. Stack scheduling does not alter the instruction sequence except by inserting stack manipulation instructions and by removing load instructions. This means it has to rely on the preceding instruction scheduling phase to yield code to be optimized. If this scheduling does not take the subsequent optimizations into account, stack scheduling might have a hard time optimizing the code and lead to suboptimal code.

Koopman's description of stack scheduling ([Koo92]) includes some figures on the efficiency of his approach. Unfortunately, they are of limited use, because only relative improvements over unoptimized code are discussed, where "unoptimized code" is not defined clearly. A second optimization technique, presented in the following section, uses instruction scheduling to generate optimal stack code for a basic block. By comparing optimal code with the results achieved by stack scheduling, we can assess the efficiency of stack scheduling and report if it could be improved by further enhancements to the basic algorithm.

[3] Koopman introduces a stack manipulation named `under` `( a b -- a a b )` for this purpose.

4.2 Optimal DAG Scheduling

4.2.1 Dependency Graphs

Directed acyclic graphs (DAGs for short) are frequently used to represent arithmetic expressions and their subexpressions ([Aho86]). In contrast to syntax trees, DAGs may contain nodes with more than one parent—those nodes indicate common subexpressions in the DAG. For our purposes, it is necessary to add some extensions to the basic features of a DAG (nodes and edges) to get a dependency graph that can be used by DAG scheduling:

- Each node is annotated with a list of variables (initially empty). Whenever the value symbolized by a node is assigned to a variable, the variable is added to the node's list. This information is required to generate the proper assignments when the graph is retransformed into a sequence of instructions.

- A global data structure is used to map variables to nodes: when the value represented by a node is assigned to a variable, the variable's slot of the map is set to reference the node. We need this data structure so that we can always find a variable's "current value".

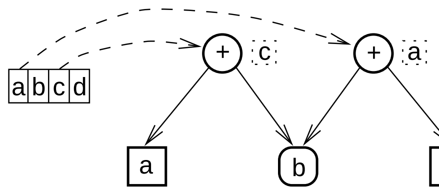
- In addition to the edges used to represent the data flow of the expression, we introduce "additional edges" to enforce a particular sequence of two nodes in the final code. Such additional edges are for example needed to correctly serialize function calls—after optimization, the functions must be called in the same order as before.

4.2.2 Scheduling

The dependency graph for a basic block is built by once again symbolically executing the instructions; this time, the stack upon which the symbolic execution takes place consists of the nodes which represent the value of the corresponding stack element at run time ([Ert92]). The global data structure described in the previous section comes in handy for suppressing unnecessary loads of local variables: each time a variable is loaded, we can first check if the graph already contains a node representing the current value of the variable that can be used instead of creating a new one. Common subexpressions in a basic block are easily spotted in a similar fashion. Before creating a node, the graph is searched for a semantically equivalent node (i.e. a node with the same operation and child nodes as the one to be created).

Example 4 shows the dependency graph for the code of example 1. The right part shows the nodes (boxes indicate leaf nodes, internal nodes are represented by circles, boxes with rounded edges are used for "shared nodes", i.e. nodes with more than one parent) and edges of the graph. The global data structure in the left part of example 4 is used to map variables to nodes as described above.

Example 4 Dependency graph for example 1



The edges of a dependency graph set up a relation that must be obeyed when retransforming the graph into a sequence of instructions. If there is an edge connecting node V_{parent} to node V_{child}, the instructions of V_{child} must precede those of node V_{parent} in the final code. Each permutation of the nodes of a graph fulfilling this condition for all nodes is called a *(valid) schedule* of the dependency graph.

In general, the number of schedules of a dependency graph can be very large (in the worst case—graphs without edges—there are $n!$ schedules of a graph with n nodes). The main problem is to find an optimal element in the set of all valid schedules. For our purposes, "optimal" means that the stack code represented by the schedule takes less or equal time to execute than the stack code represented by any other schedule without introducing new temporary variables. We calculate the "time to execute" by assuming the following timing characteristics for each instruction of a schedule:

- Accesses to local variables take three machine cycles.
- Other instructions (notably stack manipulations) take one machine cycle.

Usually, an exhaustive search for the optimum is dismissed in favor of heuristics that find a good, but not necessarily optimal solution by following some "rules of thumb" ([Smo91]). Optimal DAG scheduling does not use heuristics (partly because they turned out to be difficult to find, partly because we need to find optimal code), but some techniques are needed to make an exhaustive search more efficient:

Branch and bound The search tree is pruned when the schedules in the corresponding part of the search tree are surely worse than the best schedule found hitherto. Before the first schedule has been found, a rough estimation of the execution time is used to separate "good" from "bad" solutions.

Elimination of trees Parts of the graph that look like trees (i.e. each node has at most one parent and there are no additional edges) can be preprocessed by generating the stack instructions in a depth first postorder traversal. This approach produces optimal code for the tree that can later be inserted in the proper position when scheduling the rest of the graph.

Partitioning of the graph Some graphs consist of independent subgraphs (i.e. there are no edges between nodes in different parts). The subgraphs can be optimized separately and the code for each part is concatenated to form optimal code for the whole graph.[4]

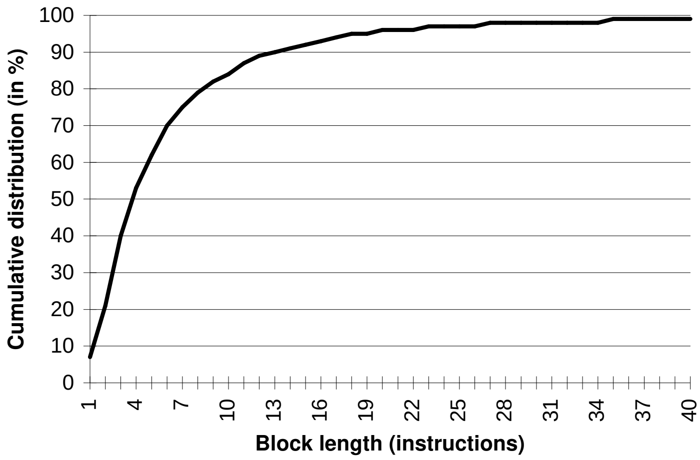
Ignoring "bad" schedules The operands to an operation represented by a node are already present in the stack (this is a consequence of a "valid" schedule). If the operands cannot be moved to the proper position for the operation by a stack manipulation, we consider the schedule "bad" and ignore it. The alternative is to save the operands in temporary variables after calculation and reload them onto the stack when needed—but this is exactly the opposite of what the optimization tries to do.

Limiting time To prevent DAG scheduling from taking too long, the user can specify a time limit. If the scheduling has not finished after that time span is over, the algorithm cannot guarantee to have found the optimal schedule. If DAG scheduling was unable to find a single valid schedule, Koopman's stack scheduling is used as fall back method to nevertheless optimize the code.

Note that it is not feasible to perform an exhaustive search for the optimal schedule of a large graph even when using the techniques described in this section. When optimizing representative JavaVM code

[4] This technique must not be applied when optimizing code for processors with instruction level parallelism, because mixing the instructions of various subgraphs could improve resource usage in the processor.

Figure 1 Length of the basic blocks in code of the JDK



(cf. section 5), we found that in a reasonable amount of time, no schedule could be found for about 1.6% of the graphs and only a part of the search tree was processed in about 2.7% of the cases (i.e. the solution could not be proved to be optimal).

5 Empirical Results

The optimizer described in section 3 has been used to gather empirical results on an extensive set of JavaVM code including the class library of the JDK 1.0.2 for Linux,[5] the classes of the Java Generic Library (JGL) 1.1,[6] and some benchmarks of a research project at Washington University.[7] The results of the optimization turned out to be very similar for all of these sources, so we (arbitrarily) use the classes of the JGL as representative JavaVM code in this section.

The code to be optimized was generated by the `javac` Java compiler of the JDK with optimization turned on. The compiler uses a simple depth first postorder instruction scheduling that had to be used by stack scheduling, because this approach does not have its own instruction scheduling (see section 4.1.3).

The length of a basic block influences the efficiency of local optimization techniques (the larger the block, the more possibilities for optimization) and their performance—especially the time needed for an exhaustive search in DAG scheduling grows exponentially with the size of basic blocks. Figure 1 shows the cumulative frequency distribution of the length of basic blocks; the data was obtained for the class library of the JDK. Most of the basic blocks are quite short: about 50% of the blocks contain no more than four instructions, blocks with at most ten instructions account for 85% of all basic blocks. Blocks with more than 30 instructions are extremely rare.

Figure 2 shows the instruction distribution of the JGL classes before and after optimization. Note that we give static instruction frequencies, i.e. we count the instructions produced by the optimizer regardless of their execution frequencies (e.g. loops). For clarity, the instructions have been classified in the following categories:

Loads/Stores. This class contains all instructions accessing local variables.

Stack manipulations. The JavaVM instructions for stack manipulation are represented by this class.

Others. This class includes all other instructions (e.g. arithmetic instructions, ...).

The first bar in figure 2 shows the code generated by `javac -O` before optimization, the second bar shows the situation after peephole optimization only, and the last two bars present the code after optimization with Koopman's stack scheduling and optimal DAG scheduling respectively. About 37% of the instructions are used to load or store local variables before optimization, while they only account for some 29% after optimization with either technique. The amount of stack manipulation instructions rises from roughly 4% of the instructions in unoptimized code to more than 13% after optimization. Table 1 summarizes the figures; it also contains an estimate of the execution time where we assume the timing characteristics described in section 4.2.2 (percentages are relative to optimal code).

The total number of instructions increases slightly after optimization, which affects performance of interpreters. Optimized code cannot be executed more efficiently in an emulated environment, because the basic requirement is not fulfilled: since every instruction

[5] `http://www.blackdown.org/java-linux.html`
[6] `http://www.objectspace.com/jgl/`
[7] `http://www.cs.washington.edu/research/interpreters/`

Figure 2 Instruction distribution in code of the JGL classes

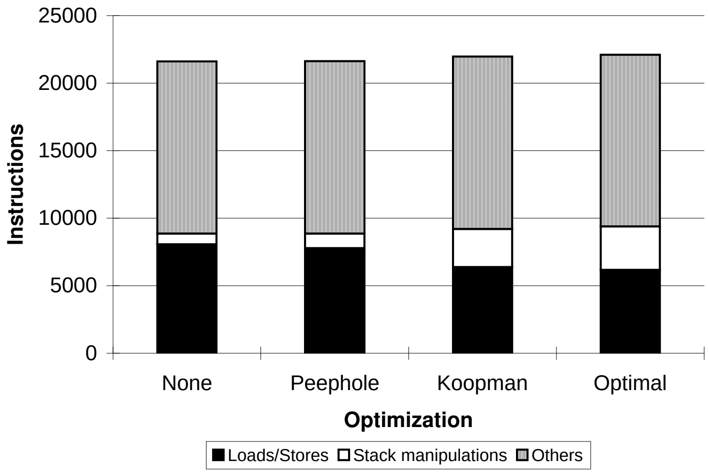


Table 1 Instruction distribution in code of the JGL classes

Optimization	None		Peephole		Koopman		Optimal	
Loads/Stores	8063	36.5%	7792	35.3%	6383	28.9%	6175	28.0%
Stack manip.	793	3.6%	1072	4.8%	2818	12.8%	3219	14.6%
Others	12761	57.8%	12761	57.8%	12761	57.8%	12703	57.4%
$\sum$	21617	97.9%	21625	97.9%	21962	99.5%	22097	100.0%
Exec. time	37743	109.6%	37209	108.0%	34728	100.8%	34447	100.0%

has to be interpreted and the stack is represented by an area in main memory, stack manipulations take as long as accesses to local variables. However, hardware implementations often have performance characteristics, that make this kind of optimization useful (i.e. main memory accesses take longer than stack manipulations).

We also gathered gathered results for an optimization that does not take commutativity of instructions like `iadd` into account and compared them with the results shown above. The variable accesses stay roughly the same, while the number of stack manipulations increases slightly when commutativity is ignored.

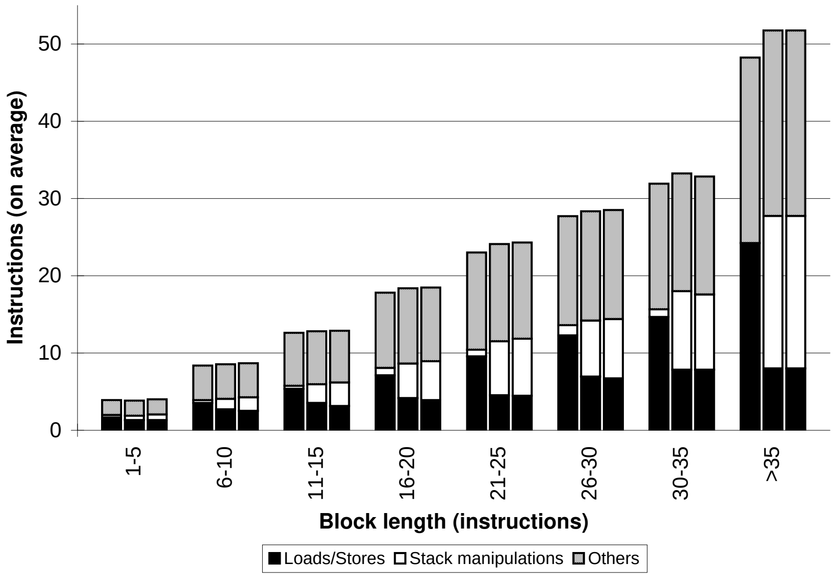
To assess the influence of the block length on the level of optimization that can be achieved, the results are broken down for varying block lengths in figure 3. The bars are grouped in sets of three (the first bar representing unoptimized code, the second one showing code optimized via stack scheduling and the last one the code after optimization by optimal DAG scheduling). For each range of block lengths, we present the instruction distribution in a similar fashion as we did in figure 2. Optimization indeed works better for longer blocks; long blocks, however, are relatively rare (cf. figure 1). Both optimization techniques perform equally well independent of the block length.

Finally, figure 4 contains data about the usage of stack instructions. Obviously, the number of stack manipulation instructions increases after optimization, but the distribution among the different instructions changes as well. Most prominently, `dup` accounts for 85% of all stack manipulations in unoptimized code, whereas optimization reduces this figure to about 60%. `swap` and `dup_x1` (the same as `tuck` in Forth) are frequently used for optimization, while the number of `pop` instructions (`drop` in Forth) does not change significantly. The `dup_x2` instruction (with a stack effect of `( a b c -- c a b c )`) is not used frequently. Because `dup`, `swap`, and `dup_x1` are most frequently used, they are "natural" candidates for efficient hardware support in future stack processors.

6 Conclusion

If stack processors are used to execute applications coded in Algol-like high level languages, new optimization techniques are needed to improve efficiency. We have presented two local optimization methods (i.e. each basic block is optimized separately) to deal with this problem. The optimizations are intended for stack processors that provide faster access to stack elements than to main memory.

Figure 3 Instruction distribution in code of the JGL classes



Both techniques try to reduce accesses to main memory that are due to loads and stores of local variables. Instead of loading a variable each time it is used, they try to keep a copy of the variable's value on the stack and use the copy in subsequent operations. Stack manipulation instructions are needed to reorder the copies for later use. It follows that the optimization we propose is only useful for stack processors, which provide faster access to the stack than to main memory.

The first optimization technique, Koopman's stack scheduling, tries to replace each load instruction in a basic block by at most two stack manipulations. The main drawback of this method is the absence of an instruction scheduling phase, i.e. the optimization has to rely on the quality of the previous instruction scheduling.

Optimal DAG scheduling, on the other hand, builds the dependency graph for a basic block and searches for an optimal schedule of its nodes (and therefore optimal code for the basic block). The exhaustive search is made more efficient by several techniques, but the time it nevertheless takes makes it unsuitable for anything but research purposes.

For representative JavaVM code, we found that stack scheduling performs very well in cooperation with a depth first postorder instruction scheduling; it produces results close to optimal code generated by DAG scheduling.

Local optimization techniques seem to be exploited with stack scheduling. Further work should concentrate on global approaches that promise a vast field of opportunities for optimization ([Koo92]).

References

[Aho86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, Reading, 1986.

[Bri94] Preston Briggs, Keith D. Cooper, Linda Torczon, Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS).* Vol. 16, No. 3, May 1994. Rice University, Houston, 1992.

[Bru75] J. L. Bruno, T. Lassagne. The Generation of Optimal Code for Stack Machines. *Journal of the ACM*, Vol. 22, No. 3, July 1975.

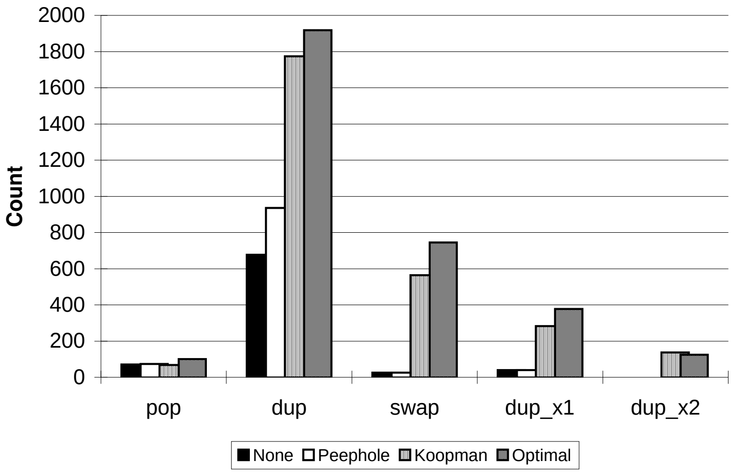
[Cha81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, Peter W. Markstein. Register Allocation Via Coloring. *Computer Languages*, Vol. 6, No. 1, 1981.

[Ert92] M. Anton Ertl. A New Approach to Forth Native Code Generation. *Proceedings of the 1992 Euroforth Conference*, Southampton, October 1992.

[Ert95a] M. Anton Ertl. Stack Caching for Interpreters. *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, La Jolla, June 1995.

[Koo89] Philip Koopman. *Stack Computers: The New Wave.* Ellis Horwood, Chichester, 1989.

Figure 4 Stack manipulations in code of the JGL classes



[Koo92] Philip Koopman. A Preliminary Exploration of Optimized Stack Code Generation. *Proceedings of the 1992 Rochester Forth Conference*, Rochester, June 1992.

[Koo93] Philip Koopman. Usenet Nuggets: Why Stack Machines? *Computer Architecture News*, Vol. 21, No. 1, March 1993.

[Mai97] Martin Maierhofer. *Erzeugung optimierten Codes für Stackmaschinen*. Diploma thesis, Vienna University of Technology, Vienna, 1997.

[Mas80] Larry M. Masinter, L. Peter Deutsch. Local Optimization in a Compiler for Stack-based Lisp Machines. *Conference Record of the 1980 LISP Conference*, Stanford, 1980. Reprint New York, 1985.

[Mil87] Daniel L. Miller. Stack Machines and Compiler Design. *Byte*, April 1987.

[Pra80] Bhaskaram Prabhala, Ravi Sethi. Efficient Computation of Expressions with Common Subexpressions. *Journal of the ACM*, Vol. 27, No. 1, January 1980.

[Smo91] Mark Smotherman, Sanjay Krishnamurthy, P. S. Aravind, David Hunnicutt. Efficient DAG Construction and Heuristic Calculation for Instruction Scheduling. *Proceedings of the 24^{th} Annual International Symposium on Microarchitecture*, Albuquerque, November 1991.

[Sun95] Sun Microsystems Computer Company. *The Java Virtual Machine Specification*,[8] 1995.

[Tan82] Andrew S. Tanenbaum, Hans van Staveren, Johan W. Stevenson. Using Peephole Optimization on Intermediate Code. *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 1, January 1982.

[Win88] A. Winfield, S. Kelly. A C-to-Forth Translator. *EuroFORML'88 Conference Proceedings*, Southampton, September 1988.

[8] `http://www.javasoft.com/doc/language_vm_specification.html`