Local Stack Allocation

Martin Maierhofer
School of Science and Technology
University of Teesside
Middlesbrough, TS1 3BA, UK
m.maierhofer@tees.ac.uk
Tel.: +44 1642 342494
Fax.: +44 1642 342401

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

Abstract

Considering the renewed interest in stack machines (in particular, the Java Virtual Machine), efficient execution of Algol-family languages on this class of hardware becomes increasingly important. Local variable accesses in the source language should be translated into stack accesses on the target machine (in analogy to register allocation on register machines).

In this paper we explore such stack allocation techniques for basic blocks. We present some improvements to Phil Koopman's *stack scheduling*, add an instruction scheduler and compare the result with an optimal stack allocation and instruction scheduling strategy. Stack scheduling in cooperation with depth first postorder instruction scheduling produces results close to the optimum. The optimizations discussed in this paper are profitable only for stack hardware where stack manipulation operations are faster than local variable accesses.

1 Introduction

After Sun introduced the Java programming language and the corresponding virtual machine (JavaVM, a stack architecture [Sun95]), the interest in stack machines has grown again. After they fell out of favor in the late seventies, stack machines were mainly used in the low-profile market of embedded control applications. Compact code size, low hardware complexity and moderate cost are the main factors responsible for their success in this area [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 perform *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 (because, in general, access to registers is much faster than access to main memory). Similarly, stack machines would profit from *stack allocation*, i.e., mapping variables to the stack.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in:
Compiler Construction (CC'98), Springer LNCS 1383, pages 189-203

Most current stack processors use a stack buffer to cache the topmost stack elements for improved performance [Koo89]. In analogy to register machines, access to these stack elements is faster than access to memory. When compiling Algol-like high level languages to stack machine code, local variables are usually usually held in the function's stack frame in main memory (similar to unoptimized code for register machines).

This situation creates an optimization opportunity: instead of loading 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.

Two properties of stack machines makes this kind of optimization more difficult than its equivalent for register machines: In contrast to register machine instructions, which (nowadays) can access any operand in any register, stack machine instructions usually have their operands implicitly in the top stack items, in a specific order. Therefore, the stack must be manipulated such that the operands appear in the right order. Moreover, stack machine instructions usually consume their operands, so if the operands are needed again later, they have to be copied first. These properties require that the compiler inserts stack manipulation instructions; one of the problems in stack allocation is to find an arrangement of stack items and a schedule for the operations such that the stack manipulation overhead is minimal.

This paper presents and evaluates local (i.e., basic block) stack allocation techniques. In [Koo92], Phil Koopman introduces a technique he calls *stack scheduling* (Section 4.1.1). Our main contribution in this paper is a meaningful evaluation of this technique: We developed an optimal scheduler and stack allocator for basic blocks (Section 4.2) and compare stack scheduling (combined with a simple instruction scheduler) to it (Section 5). We also introduce some improvements to stack scheduling (Section 4.1.2). Section 3 gives an overview of our optimizer and the related work is discussed in Section 2.

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 one common subroutine. Because calls can be executed very efficiently by stack processors, this technique reduces code size while retaining high 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 for reordering the stack elements appropriately.

Finally, [Koo92] introduces *stack scheduling*, a technique discussed in depth in Section 4.1. Stack scheduling does not reorder the code, therefore the quality of its results depends on the ordering of the original code; [Koo92] did not take this into account; the present paper does. [Koo92] evaluates stack scheduling by comparing it with extremely bad code that no compiler in its right mind would generate; we compare it with code produced by Sun's Java compiler, and with optimal code. Moreover, we also introduce a few improvements (see Section 4.1.2).

3 Optimizer Overview

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

1. First we divide the code into basic blocks and build a control-flow graph. Our basic blocks are single-entry, single-exit, as usual; however, we allow method invocations inside a basic block, because they do not alter the state of the caller's stack and local variables.

2. A live variable analysis is then performed using an iterative algorithm for 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 instructions, which reference the variable for the last time before a store to the variable or the end of the method. These instructions receive annotations about the variable becoming dead; the annotation is then used by the other optimization techniques (notably the peephole optimizer and the optimal stack allocator).

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 optimal stack allocation 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` may 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.

4 Optimization Techniques

Both approaches described in this section are "local" (i.e. each basic block is optimized separately). As a consequence, a variable needs to be used more than once in a basic block for these methods to be useful.

4.1 Stack Scheduling

4.1.1 The Basic Algorithm

Phil Koopman was the first to present a stack allocation technique—he called it *stack scheduling*[1] [Koo92]. Basically, stack scheduling substitutes loads of local variables by stack copying and manipulation instructions. Example 1 shows a short fragment of Java code in the first column. The third column shows the corresponding JavaVM code generated by `javac -O` (`a` is at index 1 in the JavaVM local variable table, `b` at index 2 and so forth).

Example 1 A fragment of Java/JavaVM code

```
c = a + b;      ( )       iload_1
                ( a )     iload_2
                ( a b )   iadd
                ( c )     istore_3   pair
a = b + d;      ( )       iload_2
                ( b )     iload_4
                ( b d )   iadd
                ( a )     istore_1
```

[1] 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." [Koo92]).

Stack scheduling starts by annotating each instruction in a basic block with information about the stack elements present at run time *before* executing the instruction (the *stack picture*). Column two in example 1 shows the annotations, with the top-of-stack being rightmost. The optimizer can determine them easily by symbolically executing the code.

Next, the algorithm tries to pair each load instruction with a preceding instruction that has a stack picture that includes the variable to be loaded. Basically, the optimizer walks through the code searching for load instructions. When it encounters a load instruction, it searches backwards, examining the stack pictures for an occurrence of the variable referenced by the load instruction. If it finds such a stack picture, it enters the corresponding instruction together with the load instruction into a list of pairs, and the algorithm continues to search for further loads.

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

In example 1, there are four load instructions. The first 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 `iload_4`, where `d` is loaded for the first and only 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 is to sort the pairs according to the distance between the two instructions. In example 1 there is not much sorting to do for a single pair. The idea behind sorting is that stack allocation is more likely to be successful for instructions that are close to one another, so they should be tried first.

The final step tries to stack-allocate the pairs[2], starting with the pairs with the smallest distance between the two instructions. A pair of instructions can be stack-allocated if the following conditions are satisfied:

- The variable can be copied to the bottom of stack by a stack manipulation right before the earlier instruction (i.e., the one, where the 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 (i.e., the load).

If a pair can be stack-allocated, the appropriate stack manipulation instruction is inserted just before the first instruction of the pair; then the second instruction (the load) 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 is just omitted). After stack-allocating a pair, the stack picture of all instructions

[2] In Koopman's terminology, *schedule* the pairs.

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 stack scheduling. Note that `dup_x1`[3] has been inserted before 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 more pairs to stack-allocate, 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 )       iload_2
                ( a b )     dup_x1
                ( b a b )   iadd
                ( b c )     istore_3
a = b + d;      ( b )       iload_4
                ( b d )     iadd
                ( a )       istore_1
```

4.1.2 Improvements

We have developed two simple extensions to the basic algorithm:

- It is not always necessary to put the copy of a variable to the bottom of stack. Instead, the copy must not be altered by an instruction executed between the first and second instruction of the pair to be stack-allocated. Because of the limited set of stack manipulation instructions provided by most stack processors, this extension enables the algorithm to stack-allocate more pairs.

- The second extension deals with searching 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 of a load is not the first one encountered during the search.

Example 3 A fragment of Java/JavaVM code

```
b = a + 5;      ( )       iload_1
                ( a )     iconst_5
                ( a 5 )   iadd
                ( b )     istore_2
c = a;          ( )       iload_1
                ( a )     istore_3
```

[3]The stack effect of `dup_x1` is `a b` $\rightarrow$ `b a b`, where the top-of-stack is rightmost.

Consider example 3. The basic algorithm would find only one pair of instructions: (iadd,iload_1). Variable a cannot be copied to the bottom of stack at the point of occurrence, so this pair cannot be stack-allocated. If we continue searching for additional partner instructions after finding the first one, another pair is found: (iconst_5,iload_1). This time, the pair can be stack-allocated (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[4].

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

4.1.3 Instruction Scheduler

When evaluating stack scheduling, we have to take the instruction scheduler into account. In the present paper, we simply used the schedule as produced by the Java compiler (javac), which appears to use a simple depth-first tree walk per statement for instruction scheduling. We also used such an instruction scheduler for a different stack machine [Mai97], with results similar to those reported here. Originally we intended to investigate more sophisticated algorithms, but the good results with the original schedule (see Section 5) convinced us that this is not necessary.

4.2 Optimal Stack Allocation

As a yardstick for evaluating the results of stack scheduling, we have implemented an optimal stack allocator and instruction scheduler. It has exponential complexity, and is therefore not suited as production optimization, but is still useful for evaluating the performance of other, more practical algorithms.

For our purposes, *optimal* code takes less or equal time to execute than any other code performing the same operations with the same data flow, without introducing 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 (in particular, stack manipulations) take one machine cycle.

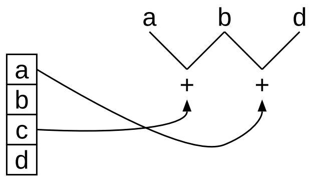
4.2.1 Dependence Graphs

For optimal stack allocation, we have to find the optimal combination of any instruction schedule and any stack allocation. Therefore our main data structure is a version of the dependence graph known from instruction scheduling.

[4] Under the assumption that stack manipulation instructions are cheaper than locals accesses.

In the dependence graph the nodes represent the computing instructions, and the edges represent ordering constraints between them; i.e., there is an edge from a to b, if a must be executed before b. Example 4 shows the dependence graph for the code of example 1.

Example 4 Dependence graph for example 1



The most common (and in our example, the only) edge type in data dependence graphs is the data flow dependence (aka true dependence, read-after-write dependence). It represents the fact that the first instruction produces data that the second instruction uses. In the JavaVM data is passed either through the stack, through a local variable, or through memory allocated in an object. There are also edges that represent other dependences: e.g., method calls have to be performed in the same order as in the original code.

In addition, our data dependence graph contains a table that maps variables to nodes and vice versa (see example 4); this table specifies which expressions reside in which variables. It is used for building the dependence graph, and for generating the assignments to local variables when the dependence graph is converted into a sequence.

4.2.2 Building the Dependence Graph

The dependence graph of a basic block is built by symbolically executing the code. 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 table mapping variables to nodes is very useful for suppressing unnecessary loads of local variables: each time a load is encountered, we can first check if the graph already contains a node representing the current value of the variable. If so, it 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).

4.2.3 Scheduling

In general, the number of schedules of a dependence graph can be very large (in the worst case—graphs without edges—there are $n!$ schedules of a graph with

n nodes). Therefore the problem is to find an optimal element in the set of all valid schedules.

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]. For optimal stack allocation we cannot use such heuristics; we use an exhaustive search, and employ the following techniques to make it run faster:

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

Elimination of trees For tree-shaped dependence graphs optimal stack allocation can be performed simply by emitting the code in a depth-first left-to-right tree walk. Therefore we can replace tree-shaped subgraphs with single nodes during our search.

Partitioning of the graph Some graphs consist of independent subgraphs (i.e. there are no edges between the subgraphs). The subgraphs can be optimized separately and the code for the parts is concatenated to form optimal code for the whole graph.[5]

Ignoring impossible schedules The operands of an operation represented by a node are already present in the stack. If the operands cannot be moved to the proper position for the operation by stack manipulation instructions, we consider the schedule impossible 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.

In the few cases where the exhaustive search takes too long in spite of these techniques, it is terminated after a user-specified time limit (per basic block). If the scheduler has not finished yet, there is no guarantee that the schedule is optimal. If optimal stack allocation was unable to find any valid schedule at all, Koopman's stack scheduling is used as fall back method.

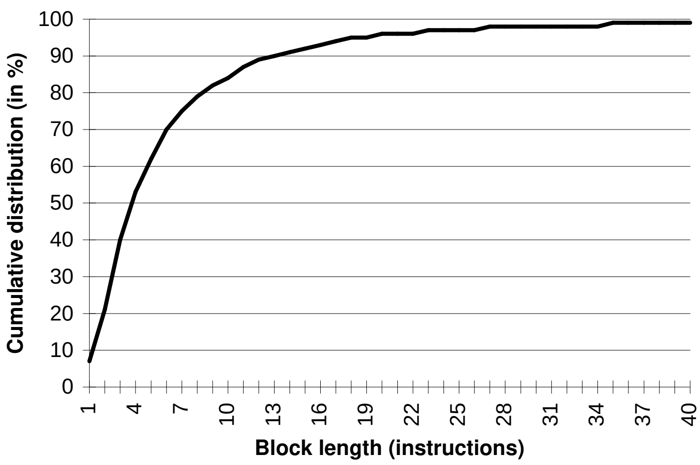
When optimizing representative JavaVM code (cf. section 5), we found that for about 1.6% of the graphs no schedule could be found in 5 seconds on a Pentium-120 and only a part of the search tree was processed in about 2.7% of the cases (i.e., the solution is not necessarily optimal).

5 Empirical Results

We have used the optimizer described in section 3 to gather empirical results on an extensive set of JavaVM code including the class library of the JDK 1.0.2 for

[5] In contrast, for register machines with instruction level parallelism the scheduler tries to mix the instructions of various subgraphs to improve resource usage in the processor.

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



Linux,[6] the classes of the Java Generic Library (JGL) 1.1,[7] and some benchmarks which are part of a research project at Washington University.[8] 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 apparently uses a simple per-statement depth-first tree-walking instruction scheduler 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 (in general, the larger the block, the more possibilities for optimization) and their performance—especially the time needed for an exhaustive search in optimal stack allocation grows exponentially with the size of basic blocks. Figure 1 shows the cumulative frequency distribution of the basic block legths 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 very rare (about 2%).

Figure 2 shows the instruction distribution in the code for the JGL classes before and after optimization. Note that we give static instruction frequencies, i.e., every basic block has the same weight, irrespective of execution frequency.

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

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, ...).

Figure 2 Instruction distribution in code of the JGL classes

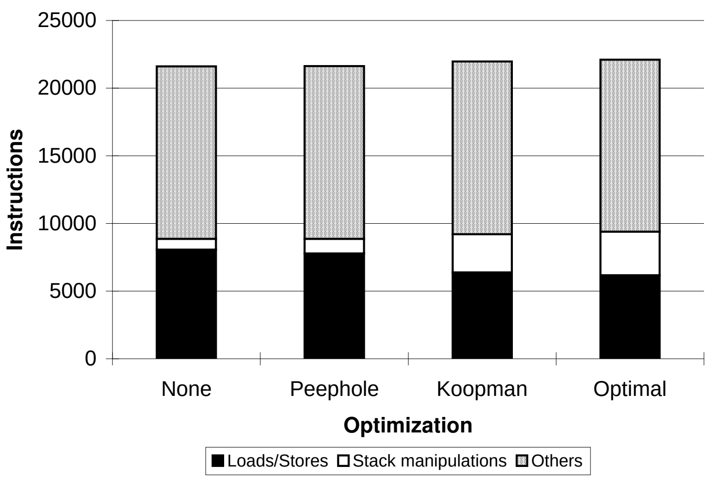


Table 1 Instruction distribution in code of the JGL classes

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%
Total	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%

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 stack allocation respectively. About 37% of the instructions are used to load or store local variables before optimization, while they only account for about 28% 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 the time for executing all basic blocks once, where we assume the timing characteristics described in section 4.2.3 (percentages are relative to optimal code).

The total number of instructions increases slightly after optimization, which means that our optimizations are only profitable if stack manipulations are cheaper than local variable accesses. I.e., it would be useful on simple processors, that have a small stack buffer, but access local variables in RAM, and also on self-timed or high-clock-rate processors, where an access to a large local variable register file is slower than dealing with the few top stack elements. Our optimizations do not pay off for implementations where local variable accesses and stack manipulations cost about the same, e.g., in an interpreter.

We also 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 stack allocation). For each range of block lengths, we present the instruction distribution in a similar fashion as 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.

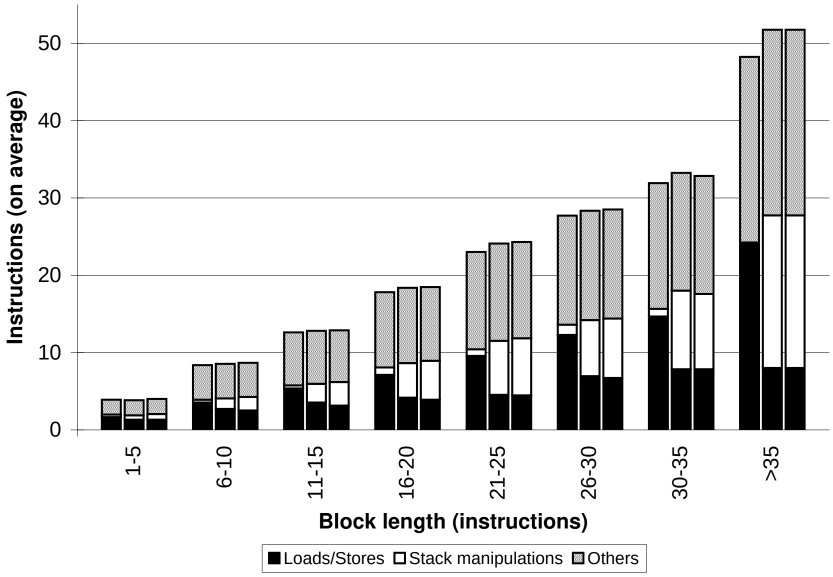
Finally, figure 4 presents results about the usage of stack instructions. Obviously, the number of stack manipulation instructions increases after optimization, but the distribution among the instructions changes as well. Most prominently, `dup` accounts for 85% of all stack manipulations in unoptimized code, whereas optimization reduces this figure to about 60%. `swap` and `dup_x1` are frequently used for optimization, while the number of `pop` instructions does not change significantly. The `dup_x2` instruction is not used frequently. Because `dup`, `swap`, and `dup_x1` are most heavily used, they are candidates for efficient hardware support in future stack processors.

6 Conclusion and Further Work

Stack allocation maps variables in source programs to the stack in the executable program for a stack machine; it is an optimization that is similar to register allocation for register machines.

In this paper we evaluate Koopman's *stack scheduling* approach to stack allocation in basic blocks in combination with an instruction scheduler by comparing it with an optimal instruction scheduler and stack allocator. Our main results are:

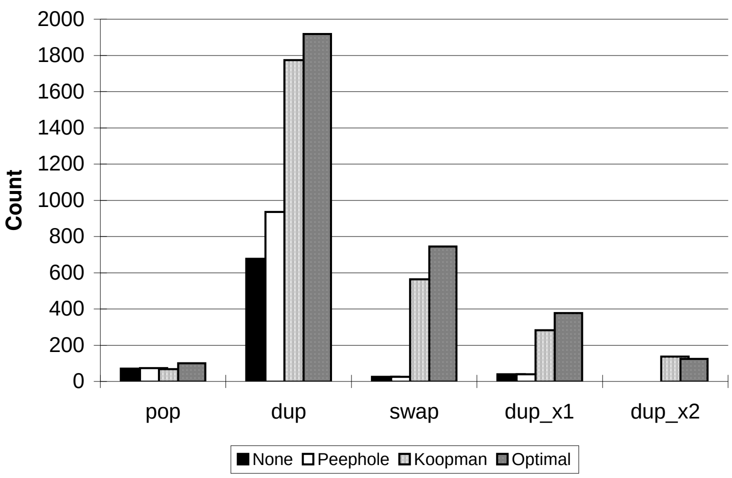
Figure 3 Instruction distribution in code of the JGL classes



- A simple depth-first post-order tree-walking instruction scheduler (as usually implemented in simple compilers) works well in combination with Koopman's stack scheduling.
- This combination produces code that is close to optimal within basic blocks for machines with high local variable access costs.
- This optimization is only profitable for machines where stack manipulations are cheap compared to local variable accesses. If both classes of instructions cost the same (e.g., on an interpreter), stack allocation *within basic blocks* is generally not profitable, because it usually replaces each local access instruction with one or more stack manipulation instruction.
- We introduce a few improvements to Koopman's stack scheduling.

Further work should concentrate on stack allocation across basic blocks, which promises many opportunities for further optimization [Koo92], and is profitable for a wider class of stack machine implementations: E.g., we stack-allocated the inner loop if the *sieve* benchmark by hand; this reduced the number of instructions in the loop from 11 (with seven local variable accesses) to 8 (with two locals accesses and two stack manipulations), giving a speedup of 20% on the whole benchmark with the JDK JavaVM interpreter.

Figure 4 Stack manipulations in code of the JGL classes



Acknowledgements

The referees provided valuable comments on the draft version of this paper. Andi Krall helped us in hand-optimizing the sieve benchmark.

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.

[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*,[9] 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.

[9] `http://java.sun.com/docs/books/vmspec/`