Treegraph-based instruction scheduling for stack-based virtual machines

Jiin Park[a,2] Jinhyung Park[a,1], Wonjoon Song[a,1], Songwook Yoon[a,1], Bernd Burgstaller[a,2], Bernhard Scholz[b,3]

[a] *Department of Computer Science*
Yonsei University
Seoul, Korea

[b] *School of Information Technologies*
The University of Sydney
Sydney, Australia

Abstract

Given the growing interest in the JVM and Microsoft's CLI as programming language implementation targets, code generation techniques for efficient stack-code are required. Compiler infrastructures such as LLVM are attractive for their highly optimizing middleend. However, LLVM's intermediate representation is register-based, and an LLVM code generator for a stack-based virtual machine needs to bridge the fundamental differences of the register and stack-based computation models.
In this paper we investigate how the semantics of a register-based IR can be mapped to stack-code. We introduce a novel program representation called treegraphs. Treegraph nodes encapsulate computations that can be represented by DFS trees. Treegraph edges manifest computations with multiple uses, which is inherently incompatible with the consuming semantics of stack-based operators. Instead of saving a multiply-used value in a temporary, our method keeps all values on the stack, which avoids costly store and load instructions. Code-generation then reduces to scheduling of treegraph nodes in the most cost-effective way.
We implemented a treegraph-based instruction scheduler for the LLVM compiler infrastructure. We provide experimental results from our implementation of an LLVM backend for TinyVM, which is an embedded systems virtual machine for C.

Keywords: bytecode instruction scheduling, stack-code, treegraphs, DAGs, LLVM

[1] Authors contributed equally.

[2] Research partially supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2010-0005234), and the OKAWA Foundation Research Grant (2009).

[3] Research partially supported by the Australian Research Council through ARC DP grant DP1096445.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

1 Introduction

Stack-code provides a compact instruction encoding, because instructions only encode the operation being performed, while its operands are implicitly consumed from the stack. Likewise, result-values of operations are implicitly produced on the stack. Network computing is facilitated by a compact program representation, because smaller programs take less time to transmit. For this reason the Java virtual machine (JVM, [12]) and various virtual machines (VMs) for wireless sensor networks, e.g., [14,19,9] employ a stack-based instruction set. Stack-based VMs are an attractive target for code generation, because several non-trivial tasks such as register allocation and callstack management are not required. All-in-all, 58 language implementations targeting the JVM and more than 52 programming languages targeting Microsoft's .NET's common language runtime (CLR, [17]) have been reported [24,23].

The renewed interest in stack-machines requires compiler support for the generation of efficient stack-code. The LLVM [11,10] compiler infrastructure is attractive for its highly-optimizing middle-end, but LLVM uses static single assignment form (SSA, [4]) as its intermediate representation (IR). Targeting LLVM to a stack-machine is complicated by the fact that SSA assumes an underlying register machine. Operands in registers can be used several times, whereas stack-based operations consume their operands.

Because the last-in, first-out stack protocol cannot support random operand access and unlimited operand lifetimes, operands that are not immediately used can be temporarily stored as local variables (temporaries) instead of keeping them on the stack. Generating bytecode to both minimize code size and improve performance by avoiding temporaries has been referred to as stack allocation [8,13]. A stack allocation method is required to provide the operands of each operation on the top of stack (TOS) when operations need them. If the underlying stack machine provides instructions to manipulate the stack, operands can be fetched from below the TOS. E.g., the JVM provides `SWAP` and `DUP` instructions to manipulate the stack. If the cost of a stack manipulation is less than the costs from stores and loads from a temporary, keeping a value on the stack is profitable.

In this paper we introduce a novel intermediate representation called treegraphs to facilitate stack allocation from register-based IRs. Treegraph nodes encapsulate computations that are compatible with a stack-based computation model. Edges between treegraph nodes represent dependencies on operands that have multiple uses. Multiple operand uses are inherently incompatible with the consuming semantics of stack-based operators. Instead of saving multiply-used values in temporaries, our method keeps all values on the stack. We perform stack allocation for treegraphs through scheduling of treegraph-nodes such that (1) dependencies between nodes are satisfied, and (2) the

number of nodes that don't require stack manipulation because their operands are already on the TOS is maximized.

The remainder of this paper is organized as follows. In Section 2 we provide background information and survey the related work. Section 3 introduces our treegraph IR and the basic treegraph scheduling algorithm. In Section 4 we discuss the minimization of overall stack manipulation costs. Section 5 contains experimental results. We draw our conclusions in Section 6.

2 Background and Related Work

Generating register-code for arithmetic expressions was first studied by Andrei Ershov [5]. Sethi and Ullman used Ershov numbers to devise an algorithm that they prove to generate optimal code for arithmetic expressions [18]. Aho and Johnson used dynamic programming to generate optimal code for expression trees on CISC machines [1].

Compiler back-ends for stack machines perform a DFS traversal of expression trees (the abstract syntax tree or other IR equivalent) and generate code as a side-effect. Common subexpression elimination finds expressions with multiple uses. The resulting DAGs complicate stack code generation: unlike values in registers, operands on the stack are consumed by their first use and are thus unavailable for subsequent uses. Fraser and Hanson's LCC [6] compiler for C comes with its own backend for a stack-based virtual machine [4]. LCC converts DAGs to trees by storing multiply-used values in temporaries and replacing references by references to the corresponding temporaries. This approach is taken with the LLVM backend for Microsoft .NET MSIL code, and it can be observed with code produced by Sun's javac compiler.

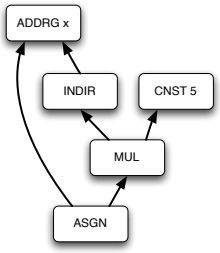
A significant amount of research has been conducted to reduce the number redundant store/load combinations by post-compilation transformations of bytecode [8,13,21,22]. In [15], dynamic instruction scheduling is performed to reduce the stack usage of a JVM. These methods are different from our approach in that they operate on the generated bytecode itself. Our approach avoids stores to temporaries altogether by keeping all values on the VM's evaluation stack.

TinyVM is a stack-based embedded systems VM for C [3]. TinyVM's instruction set closely resembles the instruction set of the bytecode backend of the LCC compiler [6]. TinyVM's instruction set is given in the appendix. Examples throughout this paper use the TinyVM instruction set.

[4] Different from the JVM

3 Treegraph IR and Treegraph Scheduling

Our optimization for the stack allocation problem is a *local* optimization, i.e., it is restricted to *single* basic blocks (BBs), where a BB is a sequence of statements with a single entry and a single exit [2].



(a)

```
1 ADDRG x
2 ADDRG x
3 INDIR
4 CNST 5
5 MUL
6 ASGN
```

(b)

```
1 ADDRL t0
2 ADDRG x
3 ASGN
4 ADDRL t0
5 ADDRL t0
6 INDIR
7 CNST 5
8 MUL
9 ASGN
```

(c)

```
1 ADDRG x
2 DUP
3 INDIR
4 CNST 5
5 MUL
6 ASGN
```

(d)

Fig. 1. Dependence-graph and instruction schedules for assignment statement `x = 5 * x`

An *acyclic* dependence graph $G(V, E, L, a)$ consists of a set of vertices V that represent instances of operations, and a set of edges $E \subseteq V \times V$ representing data dependencies among operations in V. Each each operation $o \in O$ has an arity[5] $a : O \to \mathbb{N}_0$. The mapping $L : V \to O$ assigns an operation to a vertex. In abuse of notation, we will refer to node u and $L(u)$ synonymously. A data dependency $(u, v) \in E$ is defined as a dependency between operation u and v such that operation u depends on operation v. The set of successors $S_u = \{w : (u, w) \in E\}$ of vertex u constitutes the operands of u, and they are totally ordered by relation $\prec_u \subseteq S_u \times S_u$, i.e., $v_1 \prec_u \ldots \prec_u v_k$ for successors $\{v_1, \ldots, v_k\}$. Note that the total order is given by the expected order of the operands on the stack. A dependence graph V is *well-formed* if $|S_u| = a(L(u))$, for all $u \in V$. The set P_u that denotes the predecessors of vertex u is defined similarly. We refer to operations u whose set of predecessors P_u is empty as *result* nodes of G. Figure 1(a) depicts a dependence graph for assignment statement `x=5*x`.

A stack machine can execute a sequence of instructions $\langle i_1, \ldots, i_l \rangle$ where an instruction belongs to one of the following instruction classes:

- Instruction $\texttt{op}_o$ is an instruction that performs operation $o \in O$ and has the signature $E^{a(o)} \to E$. The instruction pops $a(o)$ elements from the stack, generates one element as result, and pushes the result on top of the stack. If there are less than $a(o)$ elements on the stack, the stack machine will go into the *error state* and will terminate the execution.
- Instruction `DUP` k duplicates the top of stack element k times. Omitting argument k is equivalent to `DUP` 1. If there are no elements on the stack, the

[5] The arity is the number of operands of an operation.

stack machine will go into the error state and will terminate the execution.

- Instruction `FETCH` k duplicates element k (counted from the top) and pushes the duplicate onto the TOS. The stack size is incremented by one. If there are less than k elements on the stack, the machine will go into the error state and will terminate the execution.

A sequence of instructions $\langle i_1, \ldots, i_l \rangle$ is *admissible*, if for an empty stack none of the statements make the machine go into the error state and terminate the execution. Graph G is computed by the sequence of instructions $\langle i_1, \ldots, i_l \rangle$ iff the resulting stack contains the correct result.

Problem statement

Input: An acyclic dependence graph $G(V, E, L, a)$.
Output: code for a stack machine that performs the computations of G at minimum cost, i.e., with the minimum number of `DUP` and `FETCH` stack reordering instructions.

3.1 Special Case: G is a Tree

If G forms a tree, there exists only a single result node $r \in V$ which represents the root node of the tree. We can perform a depth-first search (DFS) as shown in Algorithm 1 where $operand(u, i)$ gives the i-th element of the total order $\prec_u$.

Algorithm 1: $DFS\,(G, u)$

```
1 foreach i ∈ {1, ..., |S_u|} do
2     DFS (G, operand(u, i))
3 emit op_{L(u)}
```

Proposition 3.1 *Sequence $DFS\,(G, r)$ is admissible and code optimal.*

Proof. The sequence is optimal because for each vertex a single instruction is emitted and this is a lower bound on the number of computations. The generated sequence is admissible and correct: This can be shown by structural induction, i.e., for each sub-tree it is shown that it is admissible and correct. The induction start are leaves of the tree. □

Note that the DFS sequence is unique in the absence of commutative operators. For a commutative operator like addition or multiplication, we get two unique trees, depending on which subtree of the commutative operator is traversed first by Algorithm 1.

Theorem 3.2 *No other instruction sequence for G is code optimal.*

Proof. The other sequence is either longer or destroys correctness. □

3.2 Code Generation for DAGs

If G is a DAG, there are no cycles but there exists a vertex v that has more than one predecessor in G.

Definition 3.3 Cut set $C \subseteq E$ is the set of all in-coming edges of nodes v which have more than one predecessor, i.e., $C = \{(u, v) \in E : |P_v| > 1\}$.

Lemma 3.4 *Graph $G(V, E - C)$ is a forest F.*

Proof. By definition every node in a forest has at most one predecessor. Hence, the lemma follows. □

The resulting forest is not well-formed in the sense that some of the operations depend on computations that are not performed inside their tree. To generate code for a DAG, we find the roots $r_1, \ldots, r_m$ for all trees $T_1, \ldots, T_m$ in F, i.e., this is the set of all nodes which have more than one predecessor in G or have no predecessors in G. We construct a tree-graph $H(F, A)$ whose set of vertices $F = \{T_1, \ldots, T_m\}$ are the trees of the forest and whose arcs $A \subseteq F \times F$ denote data dependencies between the trees, i.e., $(T_i, T_j) \in A$ iff there exists an edge $(u, v) \in E$ such that $u \in T_i$ and $v \in T_j$.

Lemma 3.5 *The tree-graph is a DAG.*

Proof. The properties of DAG G can only be destroyed by adding an edge that introduces a cycle. Condensating DAG G to a tree-graph $H(F, A)$ does not introduce additional edges, because every tree of forest F becomes a single node representing the root of the tree. The edges of the tree-graph A correspond to the cut-set C. □

For a tree $T_i \in F$ the set of successors are denoted by $\tilde{S}_{T_i}$. The successors are ordered by the depth-first-search order of T_i considering the root nodes of the successors in the order as well.

Lemma 3.6 *All trees $T_i \in F$ have either no predecessor or more than one predecessor in H.*

Proof. By construction of H. □

Algorithm 2 generates code for a treegraph H given a topological order R of H. The treegraph is traversed in reverse topological order. For every treegraph node T_i, the operands are fetched onto the TOS in reverse DFS order (lines 2–4) [6]. Then code for the tree represented by T_i is generated. The tree represented by T_i is a sub-graph of G and hence DFS gives the

[6] Fetching a value onto the TOS does not increase stack space requirements compared to storing values in temporaries, because at any one time at most one copy of a stack value is fetched to the top. Misplaced operands are popped after a treegraph has been executed, using a `POP k` instruction provided for this purpose.

Algorithm 2: Treegraph scheduler

```
1 foreach T_i ∈ F in reverse topological order R of H do
2     foreach T_j ∈ S̃_{T_i} in reverse DFS order of S̃_{T_i} do
3         if result value of T_j not in correct stack slot then
4             emit FETCH x_j
5     DFS(G, r_i);
6     if |P_{r_i}| > 0 then
7         N =Oracle (T_i, H, R) − 1;
8         if N > 0 then
9             emit DUP N
```

optimal code (line 5). Note that execution of the code generated from T_i will produce exactly one value ν on the stack unless T_i is a result node. If node T_i has predecessors (i.e., it is not a result node), then $P_{r_i} > 1$, i.e., node T_i has at least 2 predecessors (by the definition of treegraphs).

For all but the last predecessor v for which the value ν computed by T_i is in the correct stack slot, we duplicate ν (line 9). The last predecessor will consume ν itself. All other predecessors will have to fetch a copy of ν to the TOS before they are scheduled (line 4 for each of those predecessors). Clearly, the number of predecessors for which the value ν is in the correct stack slot depends on (1) the treegraph H, (2) the topological order R of H, and (3) the treenode $T_i \in H$. Algorithm 2 receives this number via an oracle (line 7). In Section 4, we will discuss our realization of this oracle via a symbolic execution of the stack state for given T_i, H, R.

As an example, consider the dependence graph of Figure 1(a). The corresponding treegraph consists of two nodes, one representing `ADDRG x`, and one representing the remaining dependence graph nodes. The reverse topological sorting for this example schedules `ADDRG x` first. The value ν produced by this treegraph node constitutes the memory address of global variable x (see the instruction set specification of TinyVM in the appendix). Value ν is already in the correct place for both uses of ν, so we duplicate once. The resulting TinyVM bytecode is depicted in Figure 1(d).

Figures 1(b) shows the bytecode where the multiply-used value is recomputed, and Figure 1(c) shows the approach where the multiply-used value is stored/loaded from a temporary variable `t0` (this is the approach chosen e.g., by the LCC compiler).

Theorem 3.7 *There exists a topological sort order R^* of H, that generates an admissible and code optimal instruction sequence.*

Proof. Local property: trees can only be generated code optimal by DFS,

i.e., for any re-ordering of instructions inside trees we can find an instruction sequence that does better (the DFS search), in which case the solution can be improved locally. □

Optimality will be achieved by a topological sort order where the number of values computed in the correct stack slot will be maximized. Section 4 will present an enumeration algorithm over all topological sortings of a treegraph.

4 Minimizing Stack Manipulation Costs

To enumerate all topological sorts of a given basic block's treegraph H, we use Knuth and Szwarcfiter's algorithm from [7]. For each topological sort, Algorithm 3 is invoked. This algorithm computes the number of `DUP` and `FETCH` instructions induced by a topological sort. The sum of the number of `DUP` and `FETCH` instructions (line 31) constitutes the cost of a topological sort. The second piece of information computed by Algorithm 3 is the oracle that tells for each treegraph node how many times the value ν computed by this node must be duplicated on the stack (the oracle was introduced with Algorithm 2).

Algorithm 3 maintains the state of the stack in variable `Stack`. When a treegraph node is about to get scheduled, the stack is consulted to check the positions of the treegraph's operands. We distinguish operations with one and two operands. Operands that are on the TOS are consumed (lines 4–5 for the two-operand case, and lines 6 and 10 for the one-operand case. For each operand not in the required position on the TOS we increase the overall counter for the required number of `FETCH` instructions by one (lines 7, 12 and 21).

To maintain the oracle-function that tells the number of times a treegraph node's result ν needs to be duplicated on the stack, the algorithm optimistically pushes one copy of ν on the stack for each predecessor. This number is saved with the oracle (lines 24–27). Every time we encounter an operand value which is not in the correct stack position, we decrement this operand's duplication counter in the oracle (lines 9, 15–16 and 23). The superfluous copy is then removed from the stack (lines 8, 13–14 and 22).

The topological sort R of minimum cost and the associated oracle are then used in Algorithm 2 to schedule treegraph nodes and emit bytecode.

We used a timeout to stop enumeration for basic blocks with too many topological sorts. For those cases the solution of minimum cost seen so far was used.

Algorithm 3: computeBasicBlockCosts

```
   Input: topsort sequence Seq of treegraph nodes
   Output: cost of Seq, Oracle[...] for number of dupped values
 1 Oracle [...] ← {}; Dup ← 0; Fetch ← 0; Stack ← empty;
 2 foreach treegraph node SU in reverse Seq do
 3     if NumberOfOperands(SU) = 2 then
 4         if Stack.top() = SU.op[1] and Stack.second() = SU.op[0] then
 5             Stack.pop(); Stack.pop()
 6         else if Stack.top() = SU.op[0] then
 7             Fetch = Fetch + 1;
 8             Stack.eraseOne(SU.op[1]);
 9             Oracle[SU.op[1]] − −;
10             Stack.pop();
11         else
12             Fetch = Fetch + 2;
13             Stack.eraseOne(SU.op[0]);
14             Stack.eraseOne(SU.op[1]);
15             Oracle[SU.op[0]] − −;
16             Oracle[SU.op[1]] − −;
17     else if NumberOfOperands(SU) = 1 then
18         if Stack.top() = SU.op[0] then
19             Stack.pop();
20         else
21             Fetch = Fetch + 1;
22             Stack.eraseOne(SU.op[0]);
23             Oracle[SU.op[0]] − −;
24     if SU computes result ν then
25         for 1 to SU.NumberOfPreds do
26             Stack.push(SU)
27         Oracle[SU] = SU.NumberOfPreds;
28 foreach treegraph node SU in Seq do
29     if Oracle[SU] > 1 then
30         Dup = Dup + 1;
31 return Dup + Fetch, Oracle[...];
```

5 Experimental Results

Figure 2 shows our experimental setup. C source code was compiled to LLVM IR by llvm-gcc ("Frontend-end"). LLVM's optimizing middle-end was by-

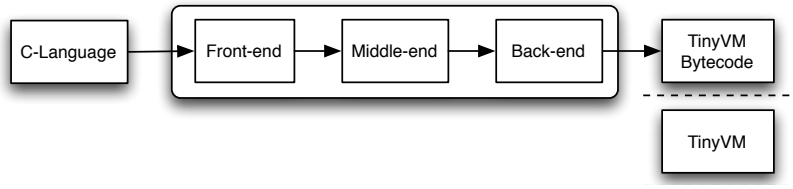


Fig. 2. Overview of the experimental setup

passed for this experiment. We implemented an LLVM backend for the generation of TinyVM bytecode from LLVM IR, based on LLVM version 2.5. The generated bytecode was then benchmarked on TinyVM.

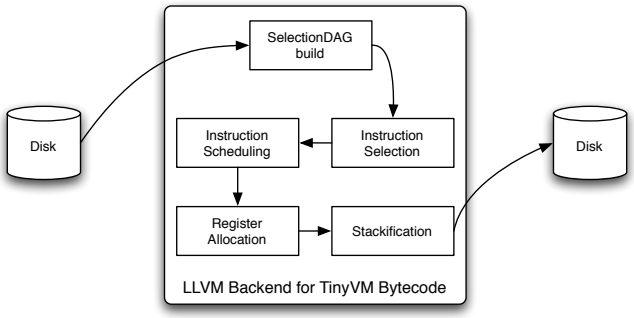


Fig. 3. LLVM Backend Structure

Figure 3 depicts the structure of our TinyVM bytecode backend. We implemented the "Instruction Selection" pass to lower LLVM IR to TinyVM bytecode instructions[7]. These bytecode instructions use pseudo-registers to accommodate SSA virtual registers. At this stage, the IR consists of DAGs where each DAG-node corresponds to one scheduable unit of bytecode instructions (i.e., a sequence of instructions that should be scheduled together). "Instruction Scheduling" denotes our treegraph instruction scheduler. For each basic block DAG we create the corresponding treegraph and run our treegraph scheduling algorithm. We use LLVM's 'local' register allocator to ensure that there are no live ranges between basic blocks. (Our optimization works on a per basic block basis; we did not consider global stack allocation yet.) Our "Stackification" pass converts pseudoregister code to stack code. This is a straight-forward conversion where operands that correspond to stack slots are dropped. For example, `ADDRG R1 x` would become `ADDRG x`. For further details we refer to [16].

All experiments were performed on an Intel Xeon 5120 server running Linux CentOS 5.4 with kernel version 2.6.18. We selected 24 C benchmark programs from the testsuite that comes with the LLVM compiler infrastruc-

[7] "Lowering" denotes the change of abstraction-level by converting LLVM IR to TinyVM bytecode.

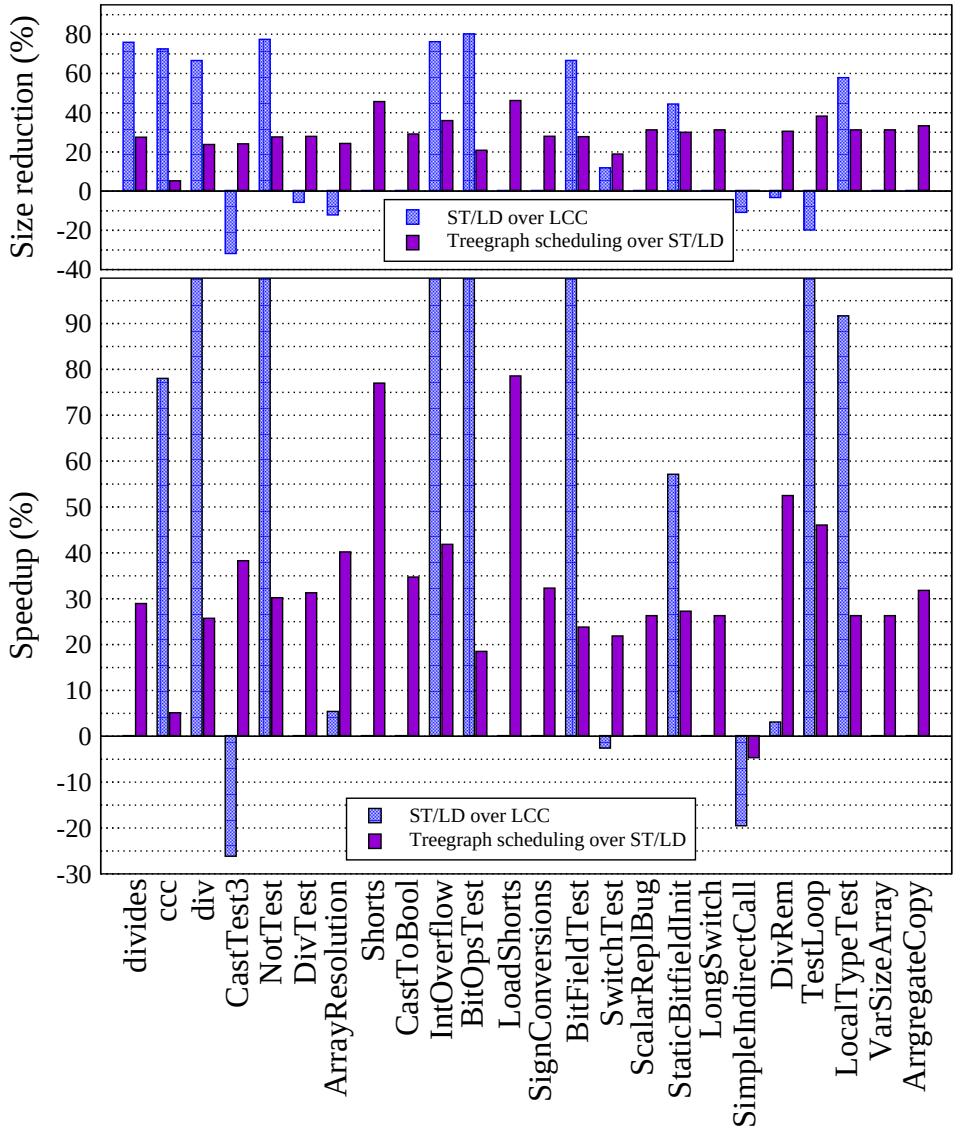


Fig. 4. Speedup and size reductions for ST/LD over LCC, and for treegraph scheduling over ST/LD. No bar with ST/LD over LCC means that the corresponding benchmark was not ANSI C and thus could not be compiled with LCC.

ture [20]. Our TinyVM backend cannot handle floats and struct args yet, which is reflected in our selection of benchmarks.

Figure 4 contains the size reductions and the speedups obtained for our benchmarks programs. Size reductions where computed as $1 - \frac{\text{newsize}}{\text{oldsize}}$.

As a yardstick for our comparison, we implemented the store/load mechanism to temporaries for our LLVM backend (i.e., every multiply-used value is written to a temporary and loaded from there when the value is required). Figure 4 depicts the improvements of our store/load mechanism over the LCC bytecode. It should be noted that LCC also applies store/load of temporaries and that the improvements are largely due to the superior code quality achiev-

able with LLVM. Note also that benchmarks with 0 improvement for ST/LD denote cases where a benchmark was not ANSI C and thus not compilable by LCC. The "Treegraph scheduling" data in Figure 4 denotes the improvement of our treegraph scheduling technique over ST/LD.

For 93% of all basic blocks our treegraph scheduler could derive the optimal solution, for the remaining 7% the enumeration of topological sorts hit the 2 second timeout. For 86% of basic blocks the solve-time was below 0.08 seconds.

6 Conclusions

In this paper we investigated how the semantics of a register-based IR can be mapped to stack-code. We introduced a novel program representation called treegraphs. Treegraph nodes encapsulate computations that can be represented by DFS trees. Treegraph edges manifest computations with multiple uses. Instead of saving a multiply-used value in a temporary, our method keeps all values on the stack, which avoids costly store and load instructions. Values that are in the correct stack slot for (some of) their users are duplicated so that they can be consumed without stack manipulation. All other values are lifted to the top of stack via a FETCH instruction. We implemented our treegraph scheduler with the LLVM compiler infrastructure for TinyVM, a stack-based embedded systems VM for C.

References

[1] Aho, A. V. and S. C. Johnson, *Optimal code generation for expression trees*, in: *STOC '75: Proceedings of seventh annual ACM symposium on Theory of computing* (1975), pp. 207–217.

[2] Aho, A. V., M. S. Lam, R. Sethi and J. D. Ullman, "Compilers: principles, techniques, and tools," Addison-Wesley, 2007, second edition.

[3] Burgstaller, B., B. Scholz and M. A. Ertl, *An Embedded Systems Programming Environment for C*, in: *Proc. Euro-Par'06* (2006), pp. 1204–1216.

[4] Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, *Efficiently computing static single assignment form and the control dependence graph*, ACM Trans. Program. Lang. Syst. **13** (1991), pp. 451–490.

[5] Ershov, A. P., *On programming of arithmetic expressions*, Communications of the ACM **1** (1958).

[6] Hanson, D. R. and C. W. Fraser, "A Retargetable C Compiler: Design and Implementation," Addison Wesley, 1995.

[7] Kalvin, A. D. and Y. L. Varol, *On the generation of all topological sortings*, Journal of Algorithms **4** (1983), pp. 150 – 162.

[8] Koopman, P. J., *A preliminary exploration of optimized stack code generation*, Journal of Forth Applications and Research **6** (1994).

[9] Koshy, J. and R. Pandey, *VMSTAR: Synthesizing Scalable Runtime Environments for Sensor Networks*, in: *SenSys '05: Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems* (2005), pp. 243–254.

[10] Lattner, C. and V. Adve, *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, in: *Proc. CGO'04* (2004).

[11] Lattner, C. A., *LLVM: an infrastructure for multi-stage optimization*, Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec (2002).

[12] Lindholm, T. and F. Yellin, "The Java Virtual Machine Specification," The Java Series, Addison Wesley Longman, Inc., 1999, second edition.

[13] Maierhofer, M. and M. Ertl, *Local stack allocation*, in: *Compiler Construction*, 1998 pp. 189–203.
URL http://dx.doi.org/10.1007/BFb0026432

[14] Müller, R., G. Alonso and D. Kossmann, *SwissQM: Next Generation Data Processing in Sensor Networks*, in: *CIDR '07: Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research* (2007), pp. 1–9.

[15] Munsil, W. and C.-J. Wang, *Reducing stack usage in Java bytecode execution*, SIGARCH Comput. Archit. News **26** (1998), pp. 7–11.

[16] Park, J., "Optimization of TinyVM Bytecode Using the LLVM Compiler Infrastructure," Master's thesis, Department of Computer Science, Yonsei University, Korea (2011).

[17] Richter, J., "CLR Via C#," Microsoft Press, 2010, third edition.

[18] Sethi, R. and J. D. Ullman, *The generation of optimal code for arithmetic expressions*, J. ACM **17** (1970), pp. 715–728.

[19] Simon, D., C. Cifuentes, D. Cleal, J. Daniels and D. White, *Java™ on the bare metal of wireless sensor devices: the squawk java virtual machine*, in: *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments* (2006), pp. 78–88.

[20] The LLVM Project, *http://llvm.org/, retrieved 2010.*

[21] Valle-Rai, R., E. Gagnon, L. Hendren, P. Lam, P. Pominville and V. Sundaresan, *Optimizing Java bytecode using the Soot framework: Is it feasible?*, in: *Compiler Construction*, LNCS **1781** (2000).

[22] Vandrunen, T., A. L. Hosking and J. Palsberg, *Reducing loads and stores in stack architectures* (2000).
URL www.cs.ucla.edu/~palsberg/draft/vandrunen-hosking-palsberg00.pdf

[23] Wikipedia, *List of CLI Languages, http://en.wikipedia.org/wiki/List_of_CLI_languages, retrieved Oct. 2010.*

[24] Wikipedia, *List of JVM Languages, http://en.wikipedia.org/wiki/List_of_JVM_languages, retrieved Oct. 2010.*

A TinyVM Instruction Set

Instruction	IS-Op.	Suffixes	Description
ADD SUB	—	FIUP..	integer addition, subtraction
MUL DIV	—	FIU...	integer multiplication, division
NEG	—	FI....	negation
BAND BOR BXOR	—	.IU...	bitwise and, or, xor
BCOM	—	.IU...	bitwise complement
LSH RSH MOD	—	.IU...	bit shifts and remainder
CNST	a	.IUP..	push literal a
ADDRG	p	...P..	push address p of global
ADDRF	l	...P..	push address of formal parameter, offset l
ADDRL	l	...P..	push address of local variable, offset l
BADDRG	index	...P..	push address of mc entity at index
INDIR	—	FIUP..	pop p; push ∗p
ASGN	—	FIUP..	pop arg; pop p; ∗p = arg
ASGN_B	a	.....B	pop q, pop p; copy the block of length a at ∗q to p
CVI	—	FIU...	convert from signed integer
CVU	—	.IUP..	convert from unsigned integer
CVF	—	FI....	convert from float
CVP	—	..U...	convert from pointer
LABEL	—	....V.	label definition
JUMP	target	....V.	unconditional jump to target
IJUMP	—	....V.	indirect jump
EQ GE GT LE LT NE	target	FIU...	compare and jump to target
ARG	—	FIUP..	top of stack is next outgoing argument
CALL	target	....V.	vm procedure call to target
ICALL	—	....V.	pop p; call procedure at p
INIT	l	....V.	allocate l stack cells for local variables
BCALL	—	FIUPVB	mc procedure call
RET	—	FIUPVB	return from procedure call
HALT	—	....V.	exit the vm interpreter
POP	k	....V.	pop k values from the TOS

Table A.1
TinyVM bytecode instruction set

Table A.1 depicts the TinyVM bytecode instruction set. The TinyVM instruction set is closely related to the bytecode interface that comes with LCC [6]. Instruction opcodes cover the leftmost column whereas the column headed "IS-Op." lists operands derived from the instruction stream (all other instruction operands come from the stack). The column entitled "Suffixes" denotes the valid type suffixes for an operand (F=float, I=signed

integer, `U`=unsigned integer, `P`=pointer, `V`=void, `B`=struct).[8] In this way instruction `ADDRG` receives its pointer argument `p` from the instruction stream and pushes it onto the stack. Instructions `ADDRF` and `ADDRL` receive an integer argument literal from the instruction stream; this literal is then used as an offset to the stack framepointer to compute the address of a formal or local variable. Unlike the JVM, TinyVM uses an `ADDR* / INDIR` instruction sequence to load a value onto the stack. To store a value, TinyVM uses the `ASGN` instruction.

[8] Operators contain *byte size* modifiers (i.e., 1, 2, 4, 8), which we have omitted for reasons of brevity.