TREEGRAPH-BASED INSTRUCTION SCHEDULING FOR STACK-BASED VIRTUAL MACHINES

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

[1]Yonsei University

[2] The University of Sydney

Benefits of stack-based VMs

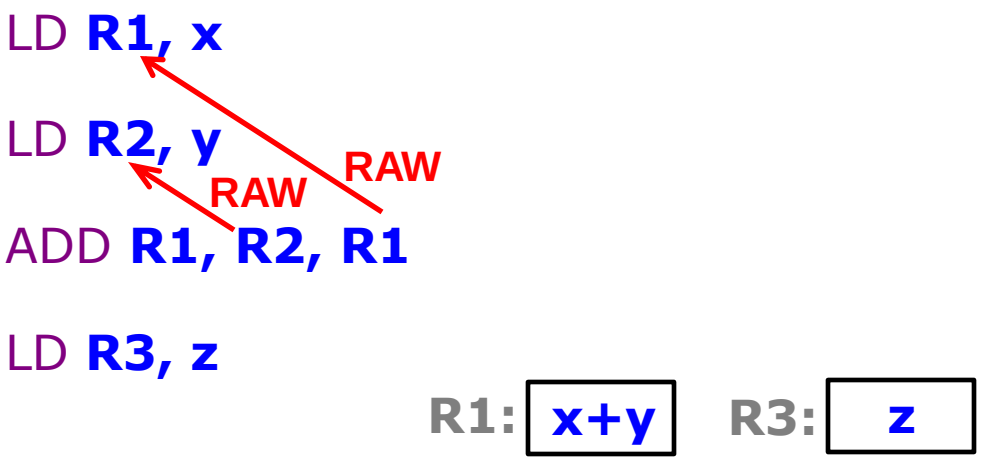
- Compact instruction encoding
  - implicit operand references to the stack
  - small programs & efficient program distribution via network
- Attractive high-level ISA
  - no register allocation nor call-stack handling required
- Languages targeting stack-based VMs:
  - >60 languages targeting the JVM
    - http://en.wikipedia.org/wiki/List_of_JVM_languages
  - >55 languages targeting Microsoft CLR
    - http://en.wikipedia.org/wiki/List_of_CLI_languages

From Register-based IR to Stack Code

- Common immediate representations (IR) used with compilers are register-based:
  - Static Single Assignment Form (SSA)
    - GCC, LLVM
  - three-address code
  - DAGs!
- To target stack-based VM, compiler backend must bridge differences between register and stack-based computation models.

Register vs. Stack Semantics

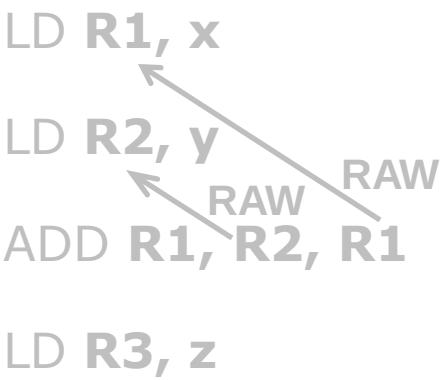
- Register-based instruction schedulers reorder instructions if data & control dependencies not violated:

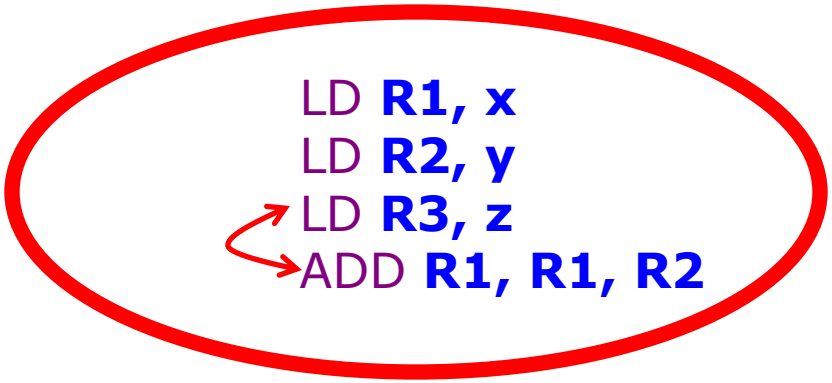


- ADD depends on preceding LD load instructions
- LD **R3, z** dependence-free
  - can go anywhere

Register Semantics & Instruction Scheduling

- Register-based instruction schedulers reorder instructions if data & control dependencies not violated:





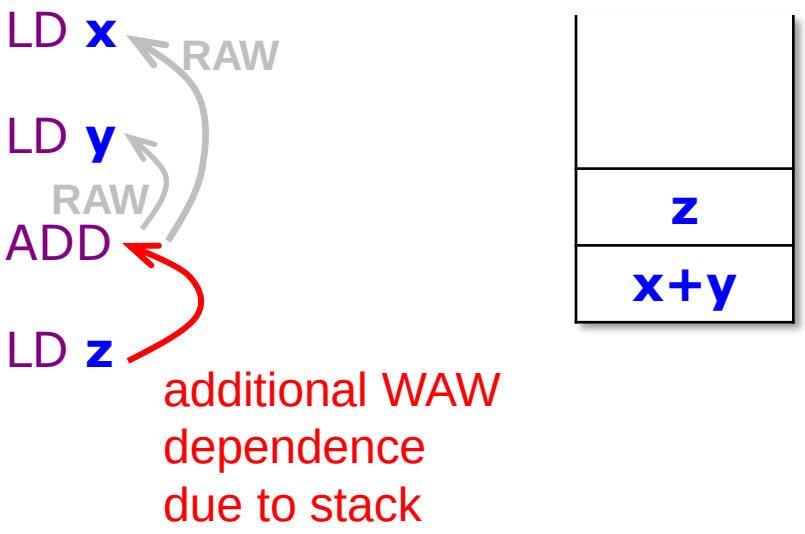
R1: x+y R3: z

- ADD depends on preceding LD load instructions
- LD **R3, z** dependence-free
  - can go anywhere
- Instruction schedulers use this property with delayed-load architectures.
  - minimizes pipeline interlocks

Stack Semantics

Stack protocol is Last-in First-out

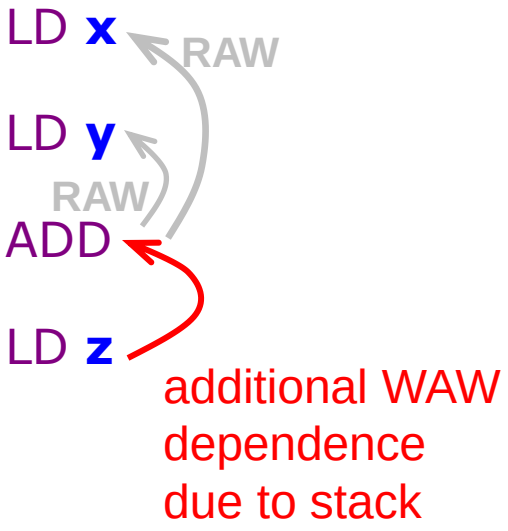
- No random operand access
- Operand access determined by stack configuration.
  - implies additional dependencies

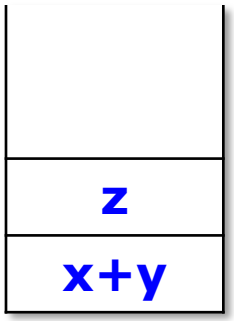


Stack Semantics

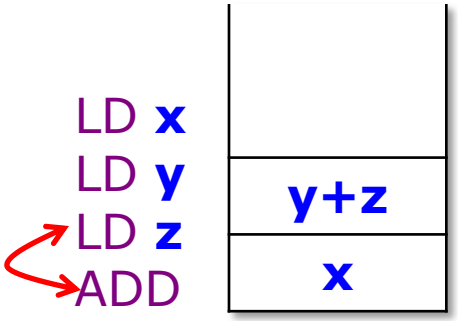
Stack protocol is Last-in First-out

- No random operand access
- Operand access determined by stack configuration.
  - implies additional dependencies





≠

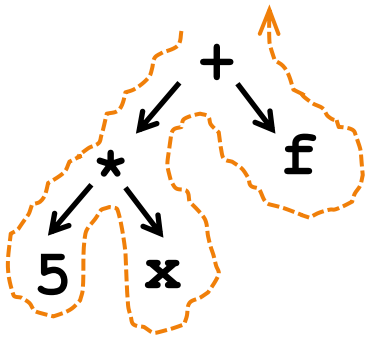


- Instruction schedulers for delayed-load architectures violate WAW dependency
- schedule 'uses' as late as possible after 'def'
- with stack: 'use' immediately after 'def'

Stack-code Generation with Trees

- With **trees**, a DFS traversal is sufficient for code generation.
  - Instructions scheduled in DFS postorder.

`5x + f`




```
1: iconst_5
2: iload_1     //LD x
3: imul
5: iload_2     //LD f
6: iadd
```

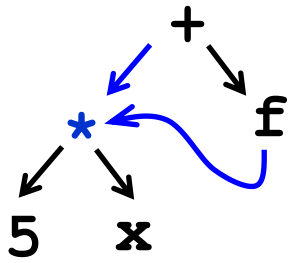
Algorithm 1: $DFS\,(G, u)$

1 foreach $i \in \{1, \ldots, |S_u|\}$ do
2 $\quad DFS\,(G, operand(u, i))$
3 emit $\mathrm{op}_{L(u)}$

Stack-code Generation with DAGS

- **DAGs** allow nodes with multiple uses
- However: stack operand use is destructive
  - operands consumed when used

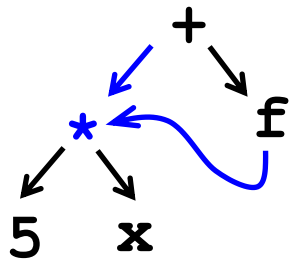
```
5x + f(5x)
```



- Code-generation approaches with DAGs (examples on next slides)
  - Recompute (e.g., openJDK javac)
  - Store value in tmp location and reload upon second use (e.g., LCC)
  - Duplicate value and keep duplicate on stack
    - [Ertl'98], [Palsberg'01]

Code-generation with DAGs


```
5x + f(5x)
```



Recompute (javac):

```
1: iconst_5
2: iload_1     //LD x
3: imul
4: iconst_5
5: iload_1     //LD x
6: imul
7: invokestatic f(I)I
8: iadd
```

Save in tmp (LCC):

```
ADDRL_P4 tmp1
CNST_I4 5
ADDRL_P4 x
INDIR_I4
MUL_I4
ASGN_I4
ADDRL_P4 tmp1
INDIR_I4
ARG_I4
ADDRL_P4 tmp2
CNST_I4 1
CALL_I4 f
...
```

tmp1=5x

LD tmp1

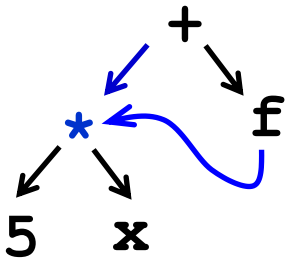
tmp2=f(tmp1)

DUP & keep on stack:

```
1: iconst_5
2: iload_1
3: imul
4: dup
5: invokestatic f(I)I
6: iadd
```

DUPlication for Multiple Uses

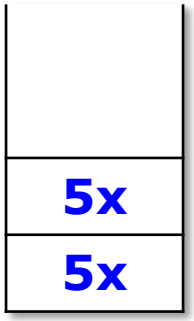
`5x + f(5x)`



DUP & keep on stack:

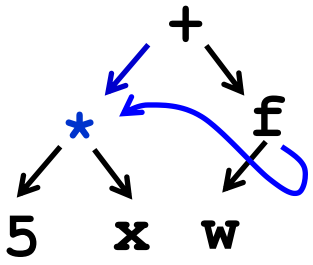
```
1: iconst_5
2: iload_1
3: imul
4: dup
5: invokestatic f(I)I
6: iadd
7: istore_1
```

Lucky case:
operand 5x duplicated
into the correct stack slot:

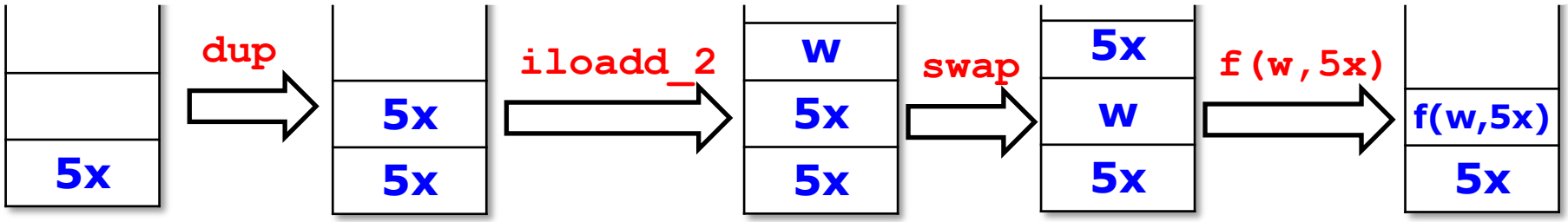


DUPlication for Multiple Uses (cont.)


```
5x + f(w, 5x)
```




```
1: iconst_5
2: iload_1      // x
3: imul
4: dup
5: iload_2      // w
6: swap
7: invokestatic f(I)I
8: iadd
9: istore_1
```



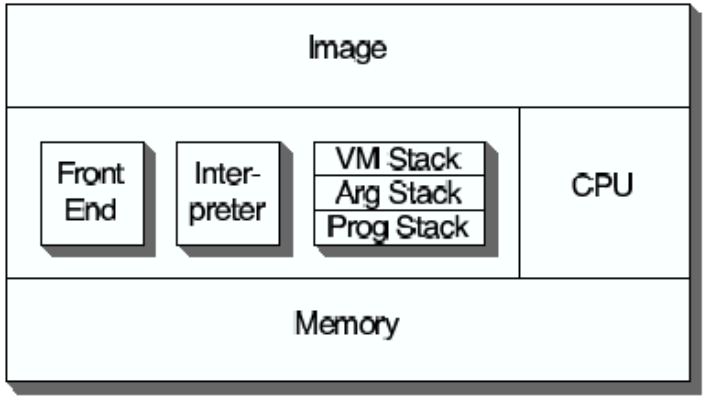
- Value not duplicated into correct stack slot:
  - stack manipulation required (swap and beyond)
    - value can be buried deeply on the stack
  - cost of manipulation must be smaller than tmp storage & recomputation!

Our contributions

- Treegraphs: a novel intermediate representation for stack allocation from register-based IRs
  - Treegraph nodes encapsulate computations compatible with stack model.
  - Edges constitute dependencies on operands with multiple uses.
- An instruction scheduling method that keeps all operands on the stack.
  - All dependencies satisfied
  - No storage of multiply-used values in tmp slots
  - #operands in correct stack slots maximized
    - to reduce stack manipulation overhead
- Experimental evaluation with the LLVM compiler infrastructure
  - for TinyVM, a stack-based embedded systems VM for C

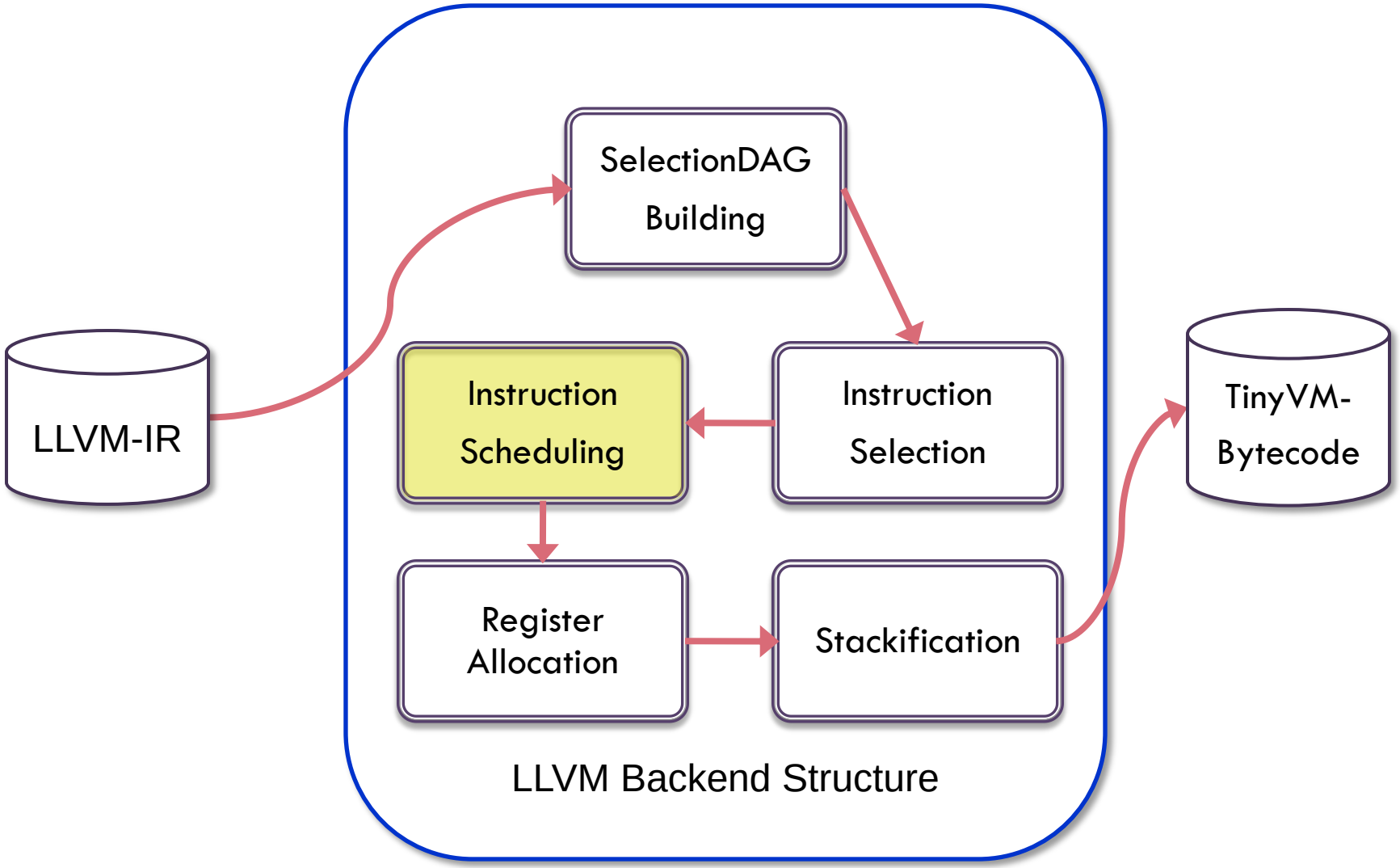
Preliminaries: TinyVM

- TinyVM is an embedded systems VM for C [EuroPar'06]
  - follows the ISA of the LCC bytecode backend [FrHa'01]
  - stack-based
  - employed vmgen VM generator [vmgen'02]
  - mixed-mode execution, code compression



- Aim: a TinyVM bytecode backend for the LLVM compiler infrastructure.

Preliminaries: LLVM TinyVM Backend



Machine Model

Stack machine executes a sequence of instructions, out of

- op: operation with a(op) being #operands.
  - The regular VM instruction set (load/store, arithmetic, logic, control-flow…)
- DUP k
  - Duplicates the top of stack (TOS) k times
  - VM stack manipulation instruction
- FETCH k
  - duplicates element k counted from top and pushes duplicate onto TOS
  - VM stack manipulation instruction

Problem Statement:

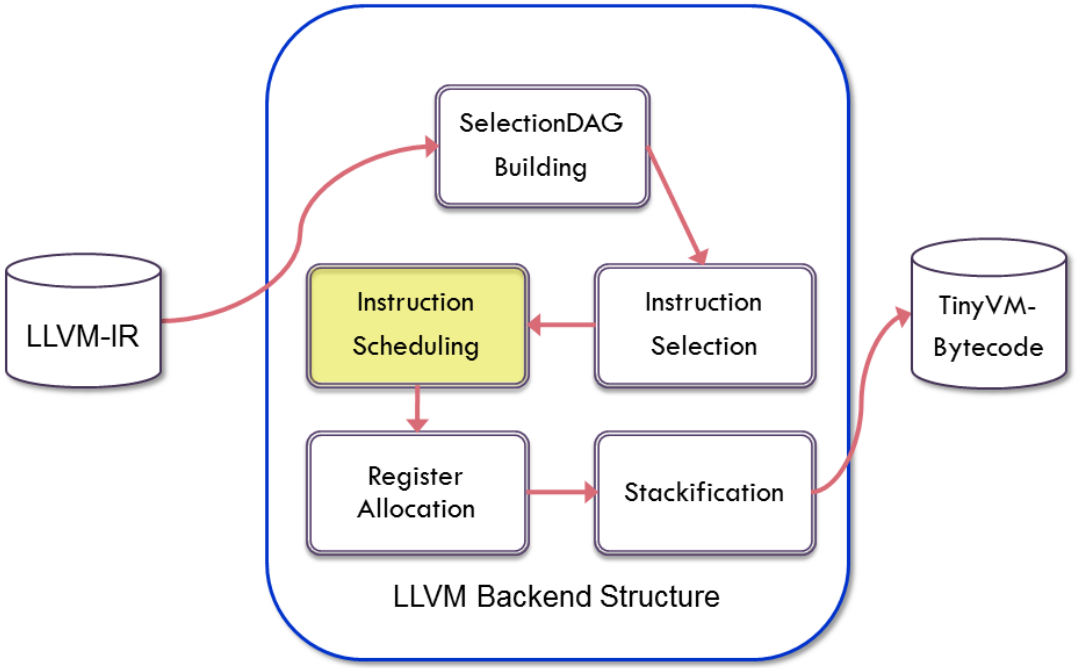
Input: acyclic dependence graph $\mathrm{G(V,E)}$

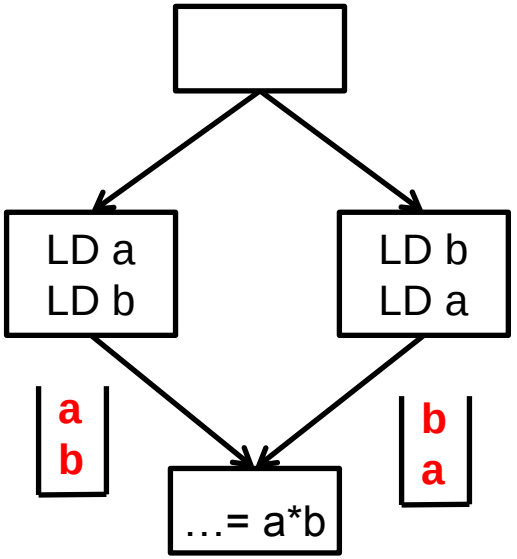
Output: code for stack machine that performs computations of G with minimum cost

Minimum cost:

- no load/store to tmp locations
- no recomputations
- minimum amount of DUP and FETCH instructions

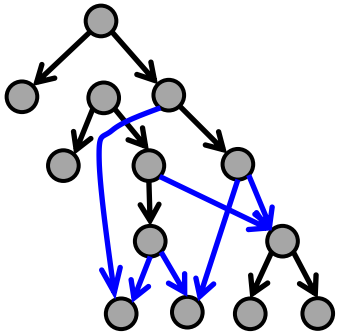
Local Instruction Scheduling



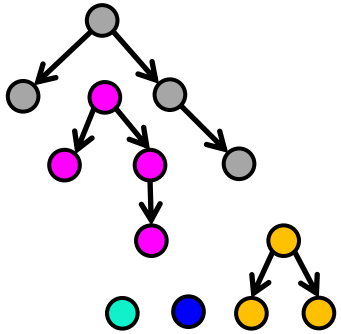


- We perform pre register-allocation scheduling
- Input in terms of LLVM: a DAG of schedulable units (SUnits)
- We use the 'local' register allocator s.t. no register is live across basic blocks (BBs).
  - → in between BBs, stack always empty.

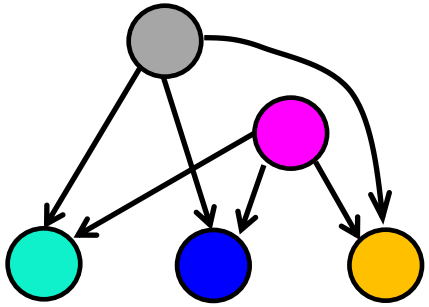
From DAGs to Treegraphs



cut set in blue



forest of trees



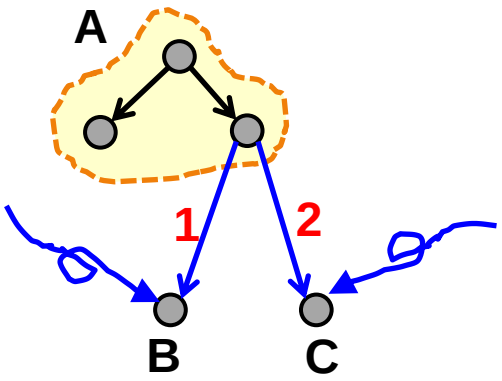
treegraph

Given a DAG:

- **Cut set**: incoming edges of DAG nodes with more than one predecessor.
- Removing the cut set partitions DAG into forest of **trees**
  - **isolates DFS-compatible treeparts of DAG**
- Trees collapsed into single nodes (treegraph nodes)
- Treegraph edges denote data dependencies between trees
  - **each edge is part of multiple operand use**

To DUP or not to DUP

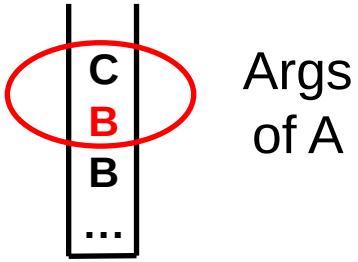
- Tree graph nodes are scheduled in reverse TOP-sort order.
- We DUPlicate only values if they are created in the correct stack slot.
  - no stack re-ordering overhead.
- Only duplicate results from treegraph nodes in DFS search order 1:



Schedule:

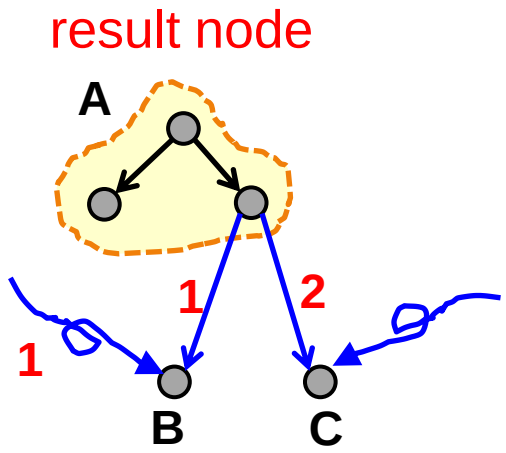
… B; DUP; C; A;…

PC



To DUP or not to DUP

- Tree graph nodes are scheduled in reverse TOP-sort order.
- We DUPlicate only values if they are created in the correct stack slot.
  - no stack re-ordering overhead.
- Only duplicate results from treegraph nodes in DFS search order 1:



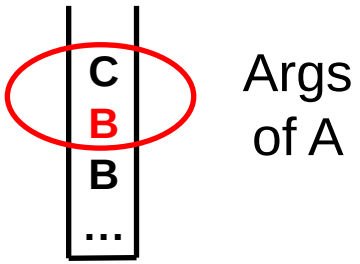
Schedule:

… B; DUP; C; A;…

PC

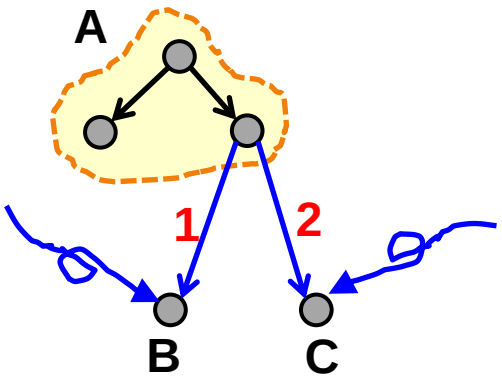
… B; C; DUP;A;…

PC



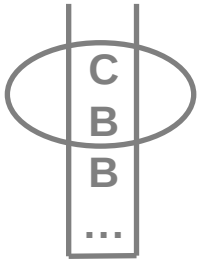
To DUP or not to DUP (cont.)

- Tree graph nodes are scheduled in reverse TOP-sort order.
- We DUPlicate only values if they are created in the correct stack slot.
  - no stack re-ordering overhead.
- Only duplicate results from treegraph nodes in DFS search order 1:



Schedule 1:

… B; DUP; C; A;…



Args of A

Schedule 2:

… B; C; DUP;A;…

DUPed **C** in the way of A's Args!

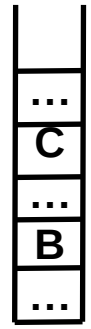
If not DUP, then FETCH

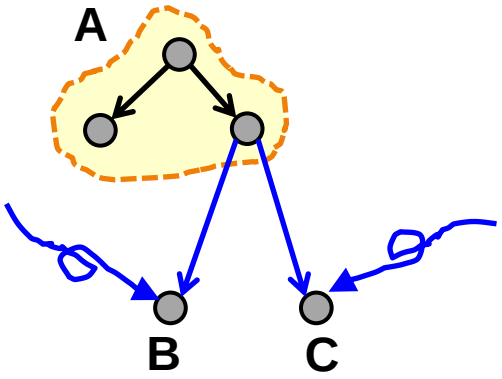
- Assume, B and C already executed due to previous uses.
- If buried on the stack, FETCH k instructions are used to copy arguments to the TOS.

Schedule:

… B; …;C;…; FETCH 4; FETCH 2;A;…

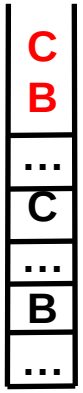
PC





… B; …;C;…; FETCH 4; FETCH 2;A;…

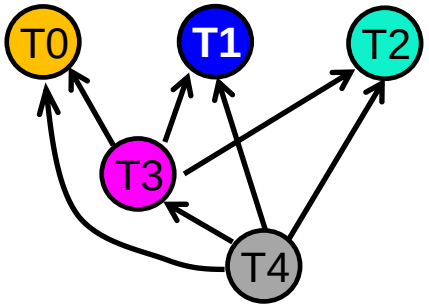
PC



Treegraph Instruction Scheduling

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
```



- After a node has been scheduled, Oracle() tells how many times the result must be duplicated (DUP)

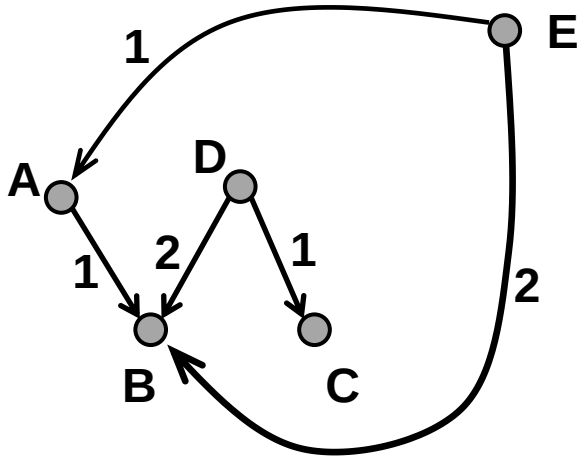
```
1  Oracle [...] ← {}; Dup ← 0; Fetch ← 0; Stack ← empty;
2  foreach treegraph node SU in reverse Seq do
3      if NumberOfOperands(SU) = 2 then
4          if Stack.top() = SU.op[1] and Stack.second() = SU.op[0] then
5              Stack.pop(); Stack.pop()
6          else if Stack.top() = SU.op[0] 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[...];
```

The Oracle

For a given schedule (which is a reverse topological sort), the Oracle…

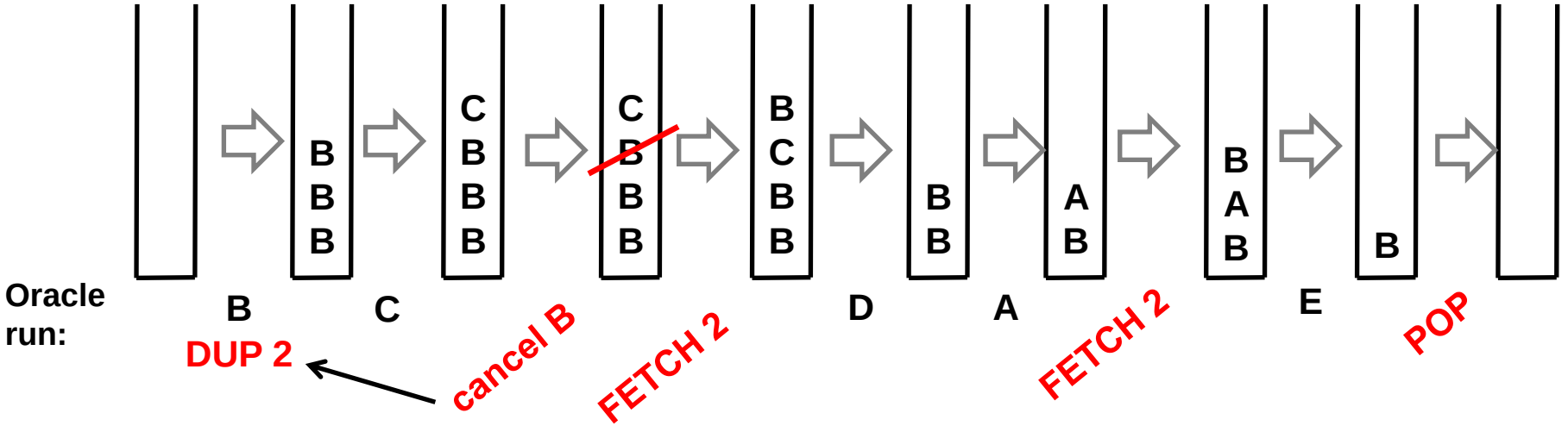
- symbolically executes the schedule
  - maintaining the stack state
- DUPs node result values optimistically
  - one for each user
- before scheduling a node, checks the stack state to determine the number of uses which are in correct slot
  - all other uses need to be fetched.

Example Oracle Run



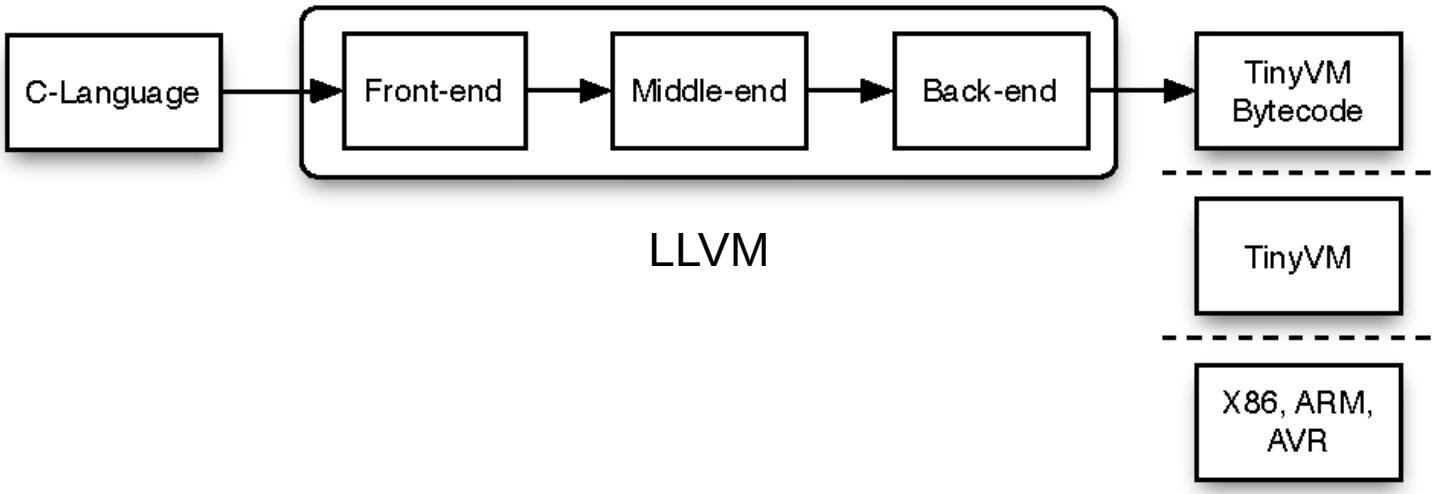
Reverse TOPsort:

B;C;D;A;E;



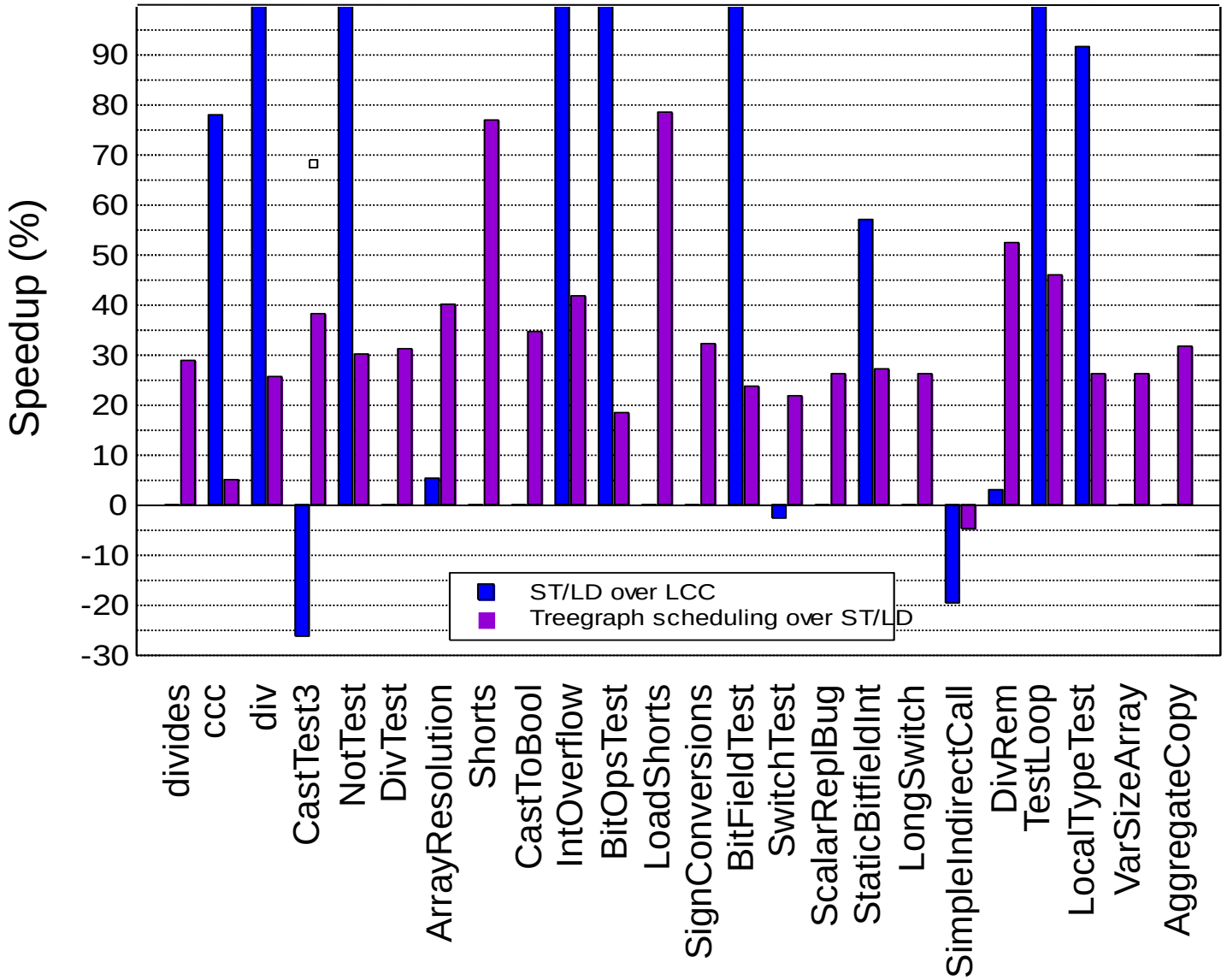
Final schedule: B; DUP; C; FETCH 2; D;A;FETCH2;E;POP;

Experimental Evaluation

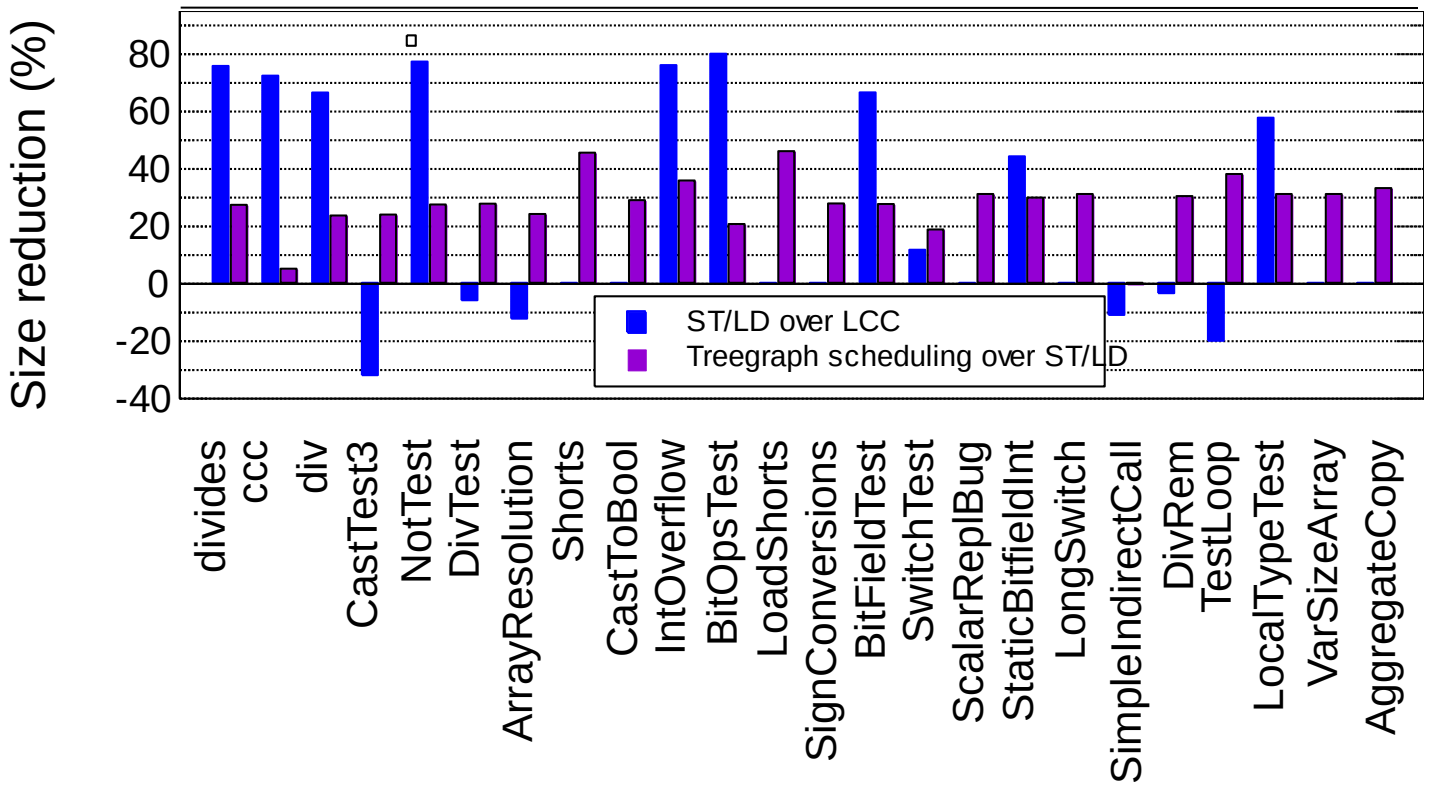


- Hardware Setup:
  - 2.33GHz dual quad-core Intel Xeon processors
    16GB memory
    Linux kernel version 2.6.23
  - Profiler uses the x86-64's hardware cycle counters
- Selected benchmarks from the LLVM benchmark suite

Speedups


Speedup (%)
90
80
70
60
50
40
30
20
10
0
-10
-20
-30
ST/LD over LCC
Treegraph scheduling over ST/LD
divides
ccc
div
CastTest3
NotTest
DivTest
ArrayResolution
Shorts
CastToBool
IntOverflow
BitOpsTest
LoadShorts
SignConversions
BitFieldTest
SwitchTest
ScalarReplBug
StaticBitfieldInt
LongSwitch
SimpleIndirectCall
DivRem
TestLoop
LocalTypeTest
VarSizeArray
AggregateCopy

Size reductions



Conclusions

- Investigated how a register-based IR can be mapped to stack-code.
- Treegraph nodes encapsulate computations that can be represented by DFS trees.
- Treegraph edges manifest computations with multiple uses.
- All values are kept on the stack.
- Values in wrong stack-slot are fetched to the TOS.
- Values in correct stack-slot are duplicated.
- Treegraph scheduler implemented for LLVM and TinyVM, a stack-based embedded systems VM for C.

Thank you…

Questions ?

Backup slides

Related Work

[1] Local Stack Allocation [Ertl‘98]

[2] Reducing Loads and Stores in Stack Architectures [Palsberg‘01]

[3] An Embedded Systems Programming Environment for C [EuroPar’06]

[4] The lcc 4.x Code-Generation Interface [FrHa’01]

[5] vmgen --- A Generator of Efficient Virtual Machine Interpreters [vmgen]

[6] Some notes on the new MLRISC X86 floating point code generator (draft).

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 p; pop arg; ∗p = arg
ASGN_B	a	.....B	pop p, pop q; 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

TinyVM Architecture

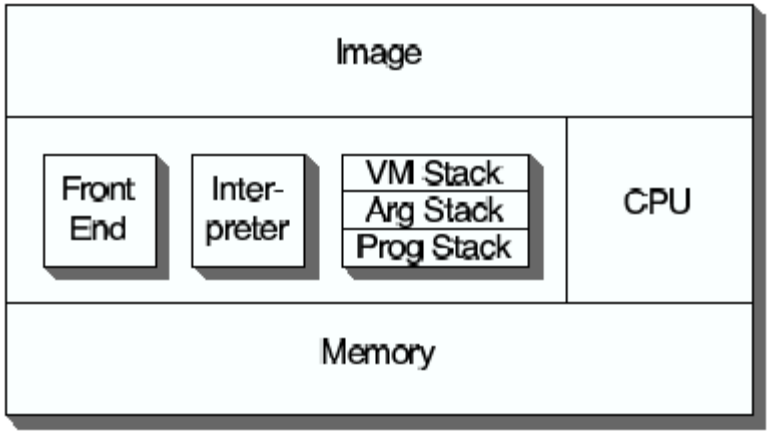


Fig. 4. Refined Execution Model

- VM Stack is the evaluation stack of the VM
- Prog stack is the stack used for machine code execution
  - (the “conventional” stack of binary programs)
- Arg stack holds procedure call arguments

Vmgen

Generated C Code:

```
LABEL(add_i4) /* add_i4 ( l1 l2 -- l ) */
{
long l1;
long l2;
long l;
NEXT_P0;
vm_Cell2l(sp[1],l1);
vm_Cell2l(spTOS,l2);
#ifdef VM_DEBUG
if (vm_debug) {
fputs(" l1=", vm_out); printarg_l(l1);
fputs(" l2=", vm_out); printarg_l(l2);
}
#endif
sp += 1;
{
l = l1+l2;
}

#ifdef VM_DEBUG
if (vm_debug) {
fputs(" -- ", vm_out); fputs(" l=", vm_out); printarg_l(l);
fputc('\n', vm_out);
}
#endif
NEXT_P1;
vm_l2Cell(l,spTOS);
NEXT_P2;
}
```

Instruction Specification:

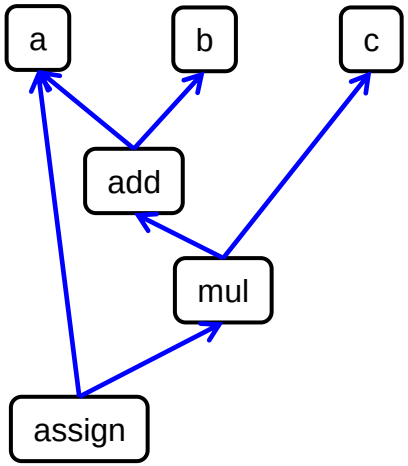
```
1 add_i4 ( l1 l2 -- l )
2 l = l1+l2;
```

(c) ADD_I4

SelectionDAG Building Pass

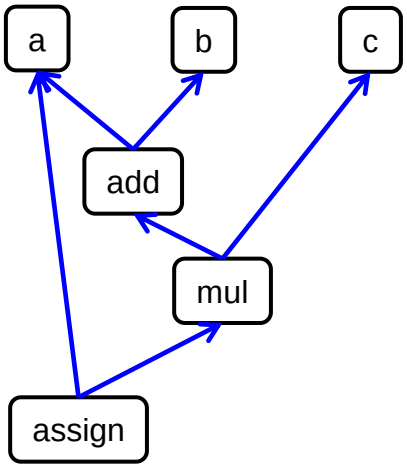
- Build DAG(Directed Acyclic Graph) from LLVM-IR

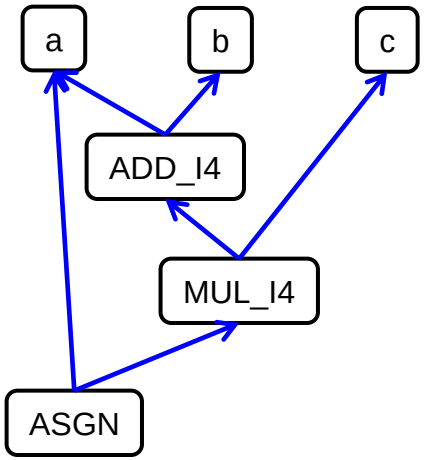

```
%1 = load i32* %a_addr
%2 = load i32* %b_addr
%3 = add i32 %1, %2
%4 = load i32* %c_addr
%5 = mul i32 %3, %4
store i32 %5, i32* %a_addr
```



Instruction Selection Pass

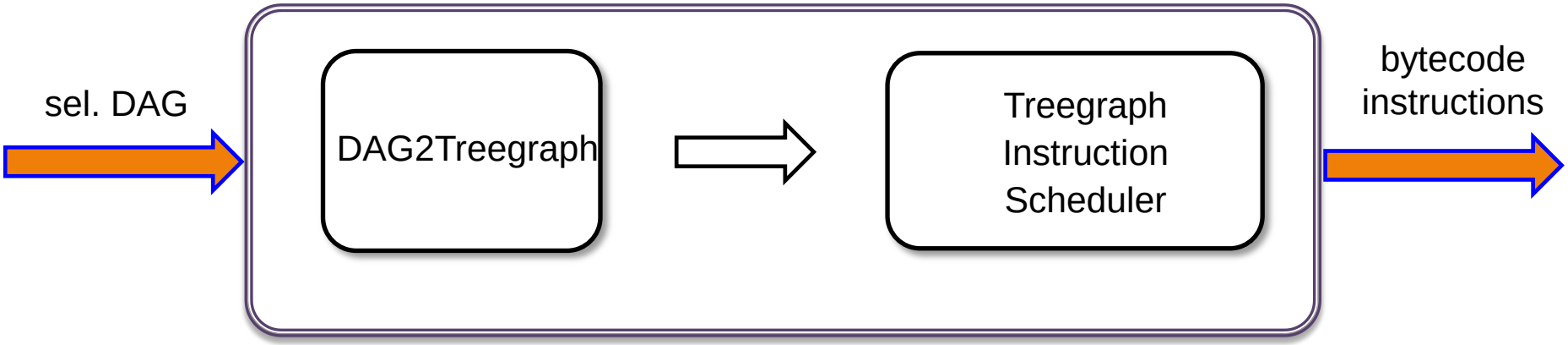
- Transform DAG nodes to instructions
  - based on pseudoregisters

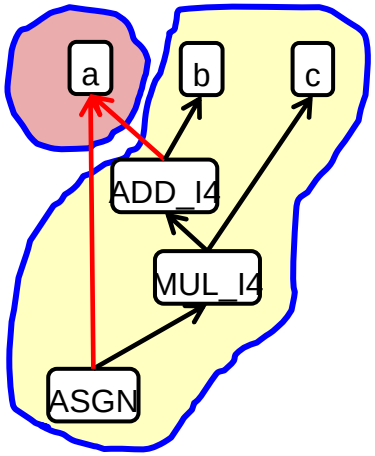


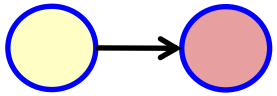


Instruction Scheduling Pass

- Convert selection dag to treegraph
- Perform instruction scheduling based on treegraph nodes







treegraph

```
%r1 = ADDG a
DUP %r1
%r2 = INDIR %r1
%r3 = ADDRG b
%r4 = INDIR %r3
%r5 = ADD %r3, %r4
...
ASGN %r1, ...
```

Register Allocation Pass

- Replace virtual registers with physical CPU registers
  - For TinyVM: not physical but **pseudo registers**

Stackification

- Converts register based code to stack based code
  - Instructions already in correct order
  - drop register references


```
%r1 = ADDRG a
DUP %r1
%r2 = INDIR %r1
%r3 = ADDRG b
%r4 = INDIR %r3
%r5 = ADD %r3, %r4
...
ASGN %r1, ...
```

```
ADDRG a
DUP
INDIR
ADDRG b
INDIR
ADD
...
ASGN
```