Avoiding Unconditional Jumps by Code Replication

by

Frank Mueller and David B. Whalley

Florida State University

Overview

- uncond. jumps
  - — occur often in programs [4-10% dynamically]
  - — produced by loops, conditional statements, etc.
  - — can almost always be avoided
- technique to avoid uncond. jumps
  - — introduction
  - — evaluation

Motivation

- code generated includes uncond. jumps for
  - while-loops and for-loops typically
  - if-then-else construct always
  - other language constructs (break, goto, continue in C)
  - as a side-effect of optimizations
- uncond. jump instruction can be avoided when code is replicated from the target
- methods often employed for certain class of loops in front-end
- new method
  - is part of optimizations in back-end of compiler
  - can be applied universally to all uncond. jumps
  - may introduce sources for other optimizations

Example 1

While-Loop (RTLs for 68020)

```
i = 1;
while (x[i]) {
    x[i-1] = x[i];
    i++;
}
```

without replication	with replication

`        a[0]=a[6]+x.+1;`
`        a[1]=a[0];`
`L15`
`        NZ=B[a[0]]?0;`
`        PC=NZ==0,L16;`
`        B[a[0]-1]=B[a[1]++];`
`        a[0]=a[0]+1;`
`        PC=L15;`
`L16     ...`
	`        NZ=B[a[6]+x.+1]?0;`
	`        PC=NZ==0,L16;`
	`        a[0]=a[6]+x.;`
	`L000`
	`        B[a[0]]=B[a[0]+1];`
	`        a[0]=a[0]+1;`
	`        NZ=B[a[0]+1]?0;`
	`        PC=NZ!=0,L000;`
	`L16     ...`

Example 2

For-Loop (RTLs for 68020)

for (i = k; i < 10; i++) x[i] = y[i];			
without replication		with replication	
	d[0]=L[a[6]+k.];		d[0]=L[a[6]+k.];
			NZ=d[0]?10;
			PC=NZ>=0,L0001;
	a[0]=d[0]+a[6]+x.;		a[0]=d[0]+a[6]+x.;
	a[1]=d[0]+a[6]+y.;		a[1]=d[0]+a[6]+y.;
	PC=L18;		
L19		L19	
	B[a[0]++]=B[a[1]++];		B[a[0]++]=B[a[1]++];
	d[0]=d[0]+1;		d[0]=d[0]+1;
L18			
	NZ=d[0]?10;		NZ=d[0]?10;
	PC=NZ<0,L19;		PC=NZ<0,L19;
	...	L0001	...

Example 3

Exit Condition in the Middle of a Loop (RTLs for 68020)

<table>
<tr><td colspan="2">i = 1;
while (i++ < n)
x[i-1] = x[i];</td></tr>
<tr><td>without replication</td><td>with replication</td></tr>
<tr><td>d[1]=1;
a[0]=a[6]+x.;
L15
d[0]=d[1];
a[0]=a[0]+1;
d[1]=d[1]+1;
NZ=d[0]?L[_n];
PC=NZ>=0,L16;
B[a[0]]=B[a[0]+1];
PC=L15;
L16 ...</td><td>d[0]=1;
d[1]=2;
NZ=d[0]?L[_n];
PC=NZ>=0,L16;
a[0]=a[6]+x.+1;
L000
B[a[0]]=B[a[0]+1];
a[0]=a[0]+1;
d[0]=d[1];
d[1]=d[1]+1;
NZ=d[0]?L[_n]
PC=NZ<0,L000;
L16 ...</td></tr>
</table>

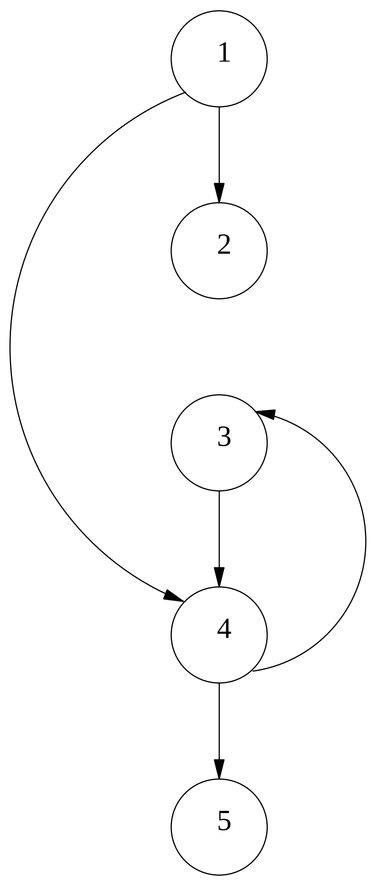
Example 4

If-Then-Else Statement (RTLs for 68020)

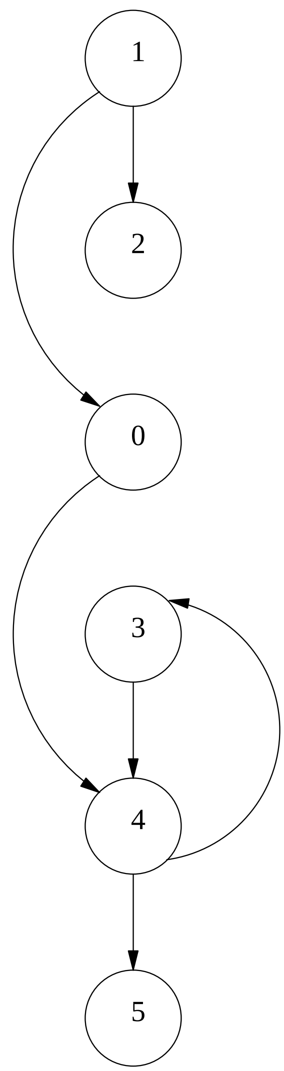
```
if (i > 5)
    i = i / n;
else
    i = i * n;
return(i);
```

without replication	with replication
NZ=L[a[6]+i.]?5;	NZ=L[a[6]+i.]?5;
PC=NZ<=0,L22;	PC=NZ<=0,L22;
d[0]=L[a[6]+i.];	d[0]=L[a[6]+i.];
d[0]=d[0]/L[a[6]+n.];	d[0]=d[0]/L[a[6]+n.];
L[a[6]+i.]=d[0];	L[a[6]+i.]=d[0];
PC=L23;	a[6]=UK;
L22	PC=RT;
d[0]=L[a[6]+i.];	L22
d[0]=d[0]*L[a[6]+n.];	d[0]=L[a[6]+i.];
L[a[6]+i.]=d[0];	d[0]=d[0]*L[a[6]+n.];
L23	L[a[6]+i.]=d[0];
a[6]=UK;	a[6]=UK;
PC=RT;	PC=RT;

Remote Preheader



No Preheader

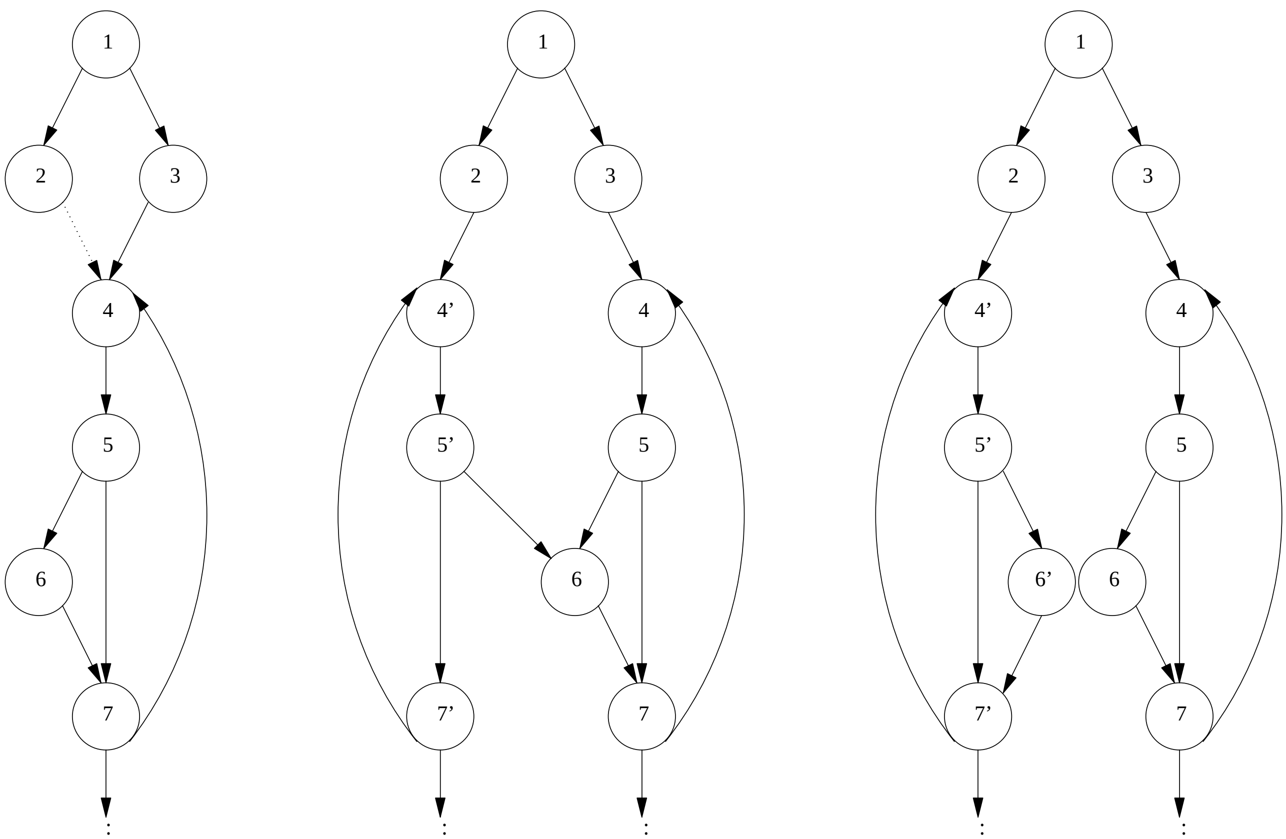


Remote Preheader

Algorithm JUMPS

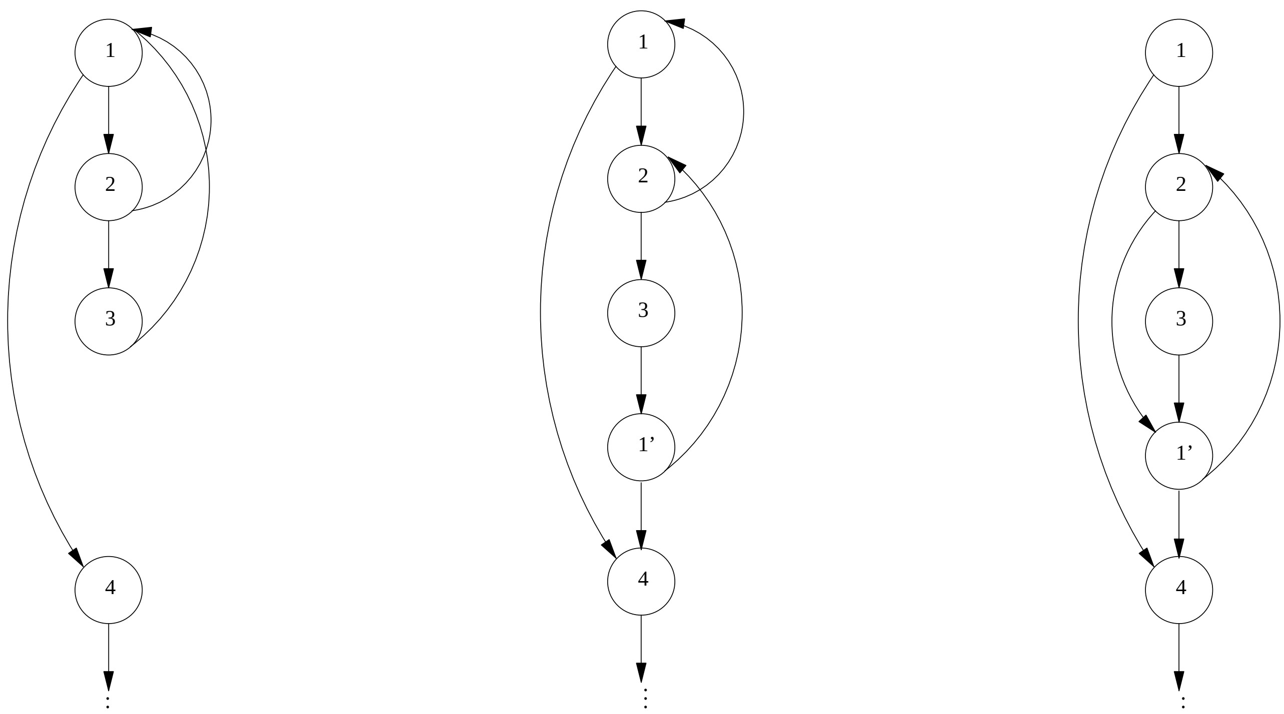
1. set up matrix (used to find shortest replication sequence)
2. traverse basic blocks until uncond. jump found
3. choose replication sequence (towards return or loop)
4. expand replication sequence to include all blocks within a loop
5. replicate code and adjust its control flow
6. adjust control flow of portion of loops which was not copied
7. if control flow has become non-reducible then remove replicated code

Interference with Natural Loops



Without Replication   With Partial Replication   With Loop Replication

Partial Overlapping of Natural Loops



Initial Control Flow After Replication Adjusted Control Flow

Measurements

- for Motorola 68020/68881 and Sun SPARC
- test programs included
  - — benchmarks
  - — UNIX utilities
  - — applications
- instrumentation of programs at code generation time
- compiled with different sets of optimizations:
  - — SIMPLE: standard opt.
  - — LOOPS: standard opt. + code replication at loops
  - — JUMPS: standard opt. + generalized code repl.

Measurements (cont.)

Number of Static and Dynamic Instructions (Sun SPARC)

Sun SPARC						
program	static instructions			dynamic instructions executed		
	SIMPLE	LOOPS	JUMPS	SIMPLE	LOOPS	JUMPS
cal	338	+3.25%	+21.89%	37,237	-2.95%	-3.15%
quicksort	321	+5.61%	+50.16%	836,404	-2.86%	-14.21%
wc	209	+0.96%	+58.37%	540,158	-0.00%	-1.96%
grep	968	+4.24%	+79.34%	1,930,791	-0.04%	-3.57%
sort	1,966	+4.63%	+89.17%	1,181,960	-0.71%	-10.49%
od	1,352	+4.59%	+95.19%	2,336,014	-8.84%	-10.22%
mincost	1,068	+6.84%	+30.99%	335,750	-0.59%	-3.91%
bubblesort	175	+7.43%	+5.14%	29,071,668	-0.05%	-0.07%
matmult	218	+4.59%	+3.67%	14,403,714	-0.08%	-0.28%
banner	169	+7.69%	+66.27%	2,565	-1.68%	-10.25%
sieve	93	+3.23%	+3.23%	2,184,965	-13.73%	-13.73%
compact	1,491	+1.07%	+75.18%	13,409,945	-1.94%	-4.86%
queens	114	+0.00%	+7.89%	263,518	-0.00%	-0.03%
deroff	7,987	+1.50%	+204.98%	448,581	-0.01%	-3.13%
average	1,176	+3.97%	+56.53%	4,784,519	-2.39%	-5.71%

Measurements (cont.)

Number of Static and Dynamic Instructions (Motorola 68020)

Motorola 68020						
program	static instructions			dynamic instructions executed		
	SIMPLE	LOOPS	JUMPS	SIMPLE	LOOPS	JUMPS
cal	323	+3.72%	+24.77%	36,290	-3.09%	-3.17%
quicksort	245	+3.67%	+37.96%	536,566	-0.39%	-3.96%
wc	173	+0.58%	+56.65%	421,038	-0.00%	-5.32%
grep	775	+3.35%	+80.90%	1,309,586	-0.03%	-3.44%
sort	1,558	+3.98%	+63.67%	902,075	-1.49%	-12.43%
od	1,198	+2.92%	+85.73%	1,980,808	-9.45%	-10.30%
mincost	906	+3.20%	+35.98%	302,062	-1.10%	-5.13%
bubblesort	137	+3.65%	+2.92%	20,340,231	-18.92%	-18.92%
matmult	146	+3.42%	+3.42%	4,891,507	-0.21%	-0.21%
banner	177	+3.95%	+55.93%	2,473	-1.42%	-13.34%
sieve	70	+1.43%	+1.43%	1,759,088	-8.53%	-8.53%
compact	1,143	+0.70%	+73.93%	10,602,159	-1.54%	-5.26%
queens	94	+0.00%	+12.77%	189,518	-0.00%	-0.05%
deroff	5,730	+1.06%	+155.17%	360,051	-0.03%	-7.05%
average	905	+2.55%	+49.37%	3,116,675	-3.30%	-6.94%

Future Work

- handle indirect jumps in algorithm
  - should improve dynamic savings
  - may reduce size of generated code
- limit length of replication sequence, use depth-bound DFS
  - should reduce size of generated code
  - should improve compile-time overhead of optimization phase
- determine best phase ordering
  - trade-off compile-time / exploit optimizations

Conclusions

- code replication
  - — avoids almost all uncond. jumps
  - — reduces number of executed instructions by 6%
  - — increases number of instructions between branches by 1.5 on SPARC
  - — results in 4% decreased cache work (except for small caches)
  - — increases code size by 53%
  - — outperforms traditional methods to avoid uncond. jumps
  - — should be applied in the back-end of highly optimizing compilers

Order of Optimizations

```
branch chaining;
dead code elimination;
reorder basic blocks to minimize jumps;
code replication (either JUMPS or LOOPS);
dead code elimination;
instruction selection;
register assignment;
if (change)
  instruction selection;
do {
  register allocation by register coloring;
  instruction selection;
  common subexpression elimination;
  dead variable elimination;
  code motion;
  strength reduction;
  recurrences;
  instruction selection;
  branch chaining;
  constant folding at conditional branches;
  code replication (either JUMPS or LOOPS);
  dead code elimination;
} while (change);
filling of delay slots for RISCs;
```