Efficient Spill Code for SDRAM

V. Krishna Nandivada and Jens Palsberg

- Processor speed is doubling every 18 months BUT memory speed is increasing only around 15% in that time.

- Every memory access is getting costlier.

- Modern processors and memory architectures allow efficient memory access.
  e.g. StrongARM + SDRAM.

Can we make the compiler aware of these properties?

1. gcc 2.95.2 compiler.

2. StrongARM processor present in the Intel's IXP-1200.

3. ILP solver as a component in the compiler.

Local scalar variables are allocated, on the stack, in such a way that

- Memory accesses are reduced.
- Memory accesses are made faster.
- Code size may not increase.
- All at the cost of affordable increase in compilation time.

Related Work

- D. Bartley - SPE 1992
- Stan Liao et.al - TOPLAS 1994
- J. Wagner and R. Leupers - LCTES 2001
- Lal George and Mathias Blume - PLDI 2003
- A. Stoutchinin
  David Goodwin and Kent D. Wilken
  Andrew Appel and Lal George

Support we get from processor and memory

- SDRAM has 64 bit bus. Loading/Storing 32 bits is inefficient.

- To use this power of SDRAM, we need a way to load/store **two** 32 bit words at a time.

- StrongARM has a load-multiple/store-multiple instruction.

StrongARM's load/store multiple instructions:

- **LDM** R_b **[list L of registers in ascending order]**
  If the list L has registers
  R_i, R_j and R_k (assume $k > j > i$)
  $R_i = [R_b]$,
  $R_j = [R_b + 4]$,
  $R_k = [R_b + 8]$

- **STM** R_b **[list of L registers in ascending order]**
  If the list L has registers
  R_i, R_j and R_k (assume $k > j > i$)
  $[R_b] = R_i$,
  $[R_b + 4] = R_j$,
  $[R_b + 8] = R_k$

ldr	$addr_1$	r_i
ldr	$addr_2$	r_j

semantically equivalent to

mov	r	$addr_1$
LDM	r	$\{r_i, r_j\}$

On SDRAM
Cost of one read (up to 64 bits) = 40 cycles.
Cost of one write (up to 64 bits) = 50 cycles.

Each LDR = 40 cycles.
Each STR = 50 cycles.

LDM to load two registers = 40 cycles !
STM to store two registers = 50 cycles !

Cost of one add/mov = 1 cycle.

Hence cost of two LDRs = 80 cycles.
Cost of add + LDM = 41 cycles!

It is advantageous to replace
two LDRs or two STRs with
an ADD followed by LDM or STM respectively.

To merge two load instructions

ldr	$addr_1$	r_i
ldr	$addr_2$	r_j

into

mov	r	$addr_1$
LDM	r	$\{r_i, r_j\}$

(If $addr_2 - addr_1 = 4$) we want $j > i$.

if $(i > j)$ then
(we call it an inversion)

- After the LDM instruction we swap the the contents of the two registers r_i and r_j.

- In case of STM, we need to swap the registers before and after the instruction.

- SDRAM allows two words to be loaded/stored at the same time.

- StrongARM has instructions to request two consecutive words to be read/written from/to memory.

	$0x10$
	$0x0C$
	$0x08$
	$0x04$
	$0x00$

- Say we have to load
  R_1 from [$0x00$]
  R_2 from [$0x04$]

  then we can combine them into one LDM.

- However say we want to load
  R_1 from [$0x00$]
  R_2 from [$0x08$]

  then we cannot merge them.

```
foo(){
  int a,b,c,d,e;
  bar1(&a,&b,&c);
  a=c+a;
  bar2(&b,&d);
  e=b+d;
  bar3(&e);
  return a+e;
}
```

```
ldr r3, [fp, #-28]   ;c
ldr r2, [fp, #-20]   ;a
add r3, r3, r2       ;a+c
str r3, [fp, #-20]   ;a
sub r0, r4
sub r1, fp, #32
bl bar2

ldr r3, [fp, #-24]   ;b
ldr r2, [fp, #-32]   ;d
add r3, r3, r2       ;b+d
str r3, [fp, #-36]   ;e
sub r0, fp, #36
bl bar3

ldr r3, [fp, #-20]   ;a
ldr r0, [fp, #-36]   ;e
add r0, r3, r0       ;a+e
```

The Problem:

Given a set of variables
we need to find a permutation
of the variables such that
the loads and stores are optimized.

The Placement Problem

Given a set of blocks of memory accesses,
find a placement function that leads to
maximizing the number of
double loads and double stores,
while minimizing the number of inversions.

A decision version of the placement problem is
NP-Complete.
(Reduce it from Hamiltonian path problem.)

SLA - Stack Location Allocator

Model Extraction

- Find all the candidate pairs (called edges) of word load/stores.

- For each edge, note the static execution frequency w.

- For each edge e note the cycle count:
  cost[e]=40 if it consists of a pair of loads and
  cost[e]=50, if it consists of a pair of stores.

This information is output to a Integer Linear Program (ILP) solver to solve.

Constraint Generation:

In the constraints generated

- Set of variables vars : $\{1..n\}$.

- Placement function
  f:array{vars $\times$ vars} of {0,1}.
  If $f[v, p] = 1$ then variable v has position p.

- For any edge e, if ILP decides to introduce a LDM/STM then isPair[e]=1.

The ILP solver needs linear constraints and a linear objective function.

• Solving the placement problem contributes to saving cycles in the overall execution.

• Our ILP maximizes an objective function which approximates the number of saved cycles.

– If two LDRs/STRs are replaced by a LDM/STM then we save around 40/50 cycles.

– Inversion $\Rightarrow$ swap registers $\Rightarrow$ saving reduced by 3/6 cycles.

$$s[e] = \text{isPair}[e] \times w[e] \times \text{cost}[e]$$

$$\text{Objective function:} \sum s[e]$$

f is a permutation matrix:

$$\forall v \in \text{vars} : \sum_{p \in vars} f[v,p] = 1$$

$$\forall p \in \text{vars} : \sum_{v \in vars} f[v,p] = 1$$

Some other constraints :

- isPair[e] can be set 1, only if e is a valid edge.

- if $f[v_1,p_1] = 1$ and $f[v_2,p_2] = 1$ then diff[v_1,v_2] $= p_2 - p_1$.

- if isPair[e] $= 1$ then $|\text{diff}[e]| = 1$.

Constraint Solving

- AMPL to frame the constraints and objective function.

- CPLEX to solve them.

Code Transformation

1. Read back the solution.

2. For each instruction do:

   - $(r = fp + offset) \longrightarrow (r = fp + offset')$
   - Modify all load/store instructions which have frame pointer as the base register.

3. For each edge e with isPair[e]=1

   - Replace the two instructions by LDM/STM.
   - Insert an add instruction.
   - Insert exclusive-or (eor) instructions if required.

```
foo(){ int a,b,c,d,e;
       bar1(&a,&b,&c);
       a=c+a;
       bar2(&b,&d);
       e=b+d;
       bar3(&e);
       return a+e;
}
```

var	old loc	new loc
a	fp-20	fp-24
b	fp-24	fp-32
c	fp-28	fp-20
d	fp-32	fp-36
e	fp-36	fp-28

Without SLA	**With SLA**	
ldr r3, [fp, #-28]	sub r2, fp, #24	
ldr r2, [fp, #-20]	ldmia r2, {r2,r3}	;**load a,c**
add r3, r3, r2	add r3, r3, r2	;*a=a+c*
str r3, [fp, #-20]	str r3, [fp, #-24]	;*store a.*
sub r0, r4	mov r0, r4	
sub r1, fp, #32	sub r1, fp, #36	;&d
bl bar2	bl bar2	;*call*
ldr r3, [fp, #-24]		
ldr r2, [fp, #-32]	ldmia sp, {r2,r3}	;**load b,d**
add r3, r3, r2	add r3, r3, r2	;*e=b+d*
str r3, [fp, #-36]	str r3, [fp, #-28]	;*store e*
sub r0, fp, #36	sub r0, fp, #28	;&e
bl bar3	bl bar3	;*call*
ldr r3, [fp, #-20]	sub r0, fp, #28	
ldr r0, [fp, #-36]	ldmia r0, {r0,r3}	;**load a,e**
add r0, r3, r0	add r0, r3, r0	;*a+e*

- MediaBench: GSM and EPIC
- NetBench: Url, Md5 and IPChains
- Purdue: Classifer and Firewall

Benchmark characteristics		
Benchmark	#funcs	#lines
GSM	98	8643
EPIC	49	3540
Url	12	790
Md5	17	753
IPChains	76	3453
Classifier	25	2850
Firewall	30	2281

Compile time statistics						
	Compile time (sec)			Xformations		
Bench	w/o SLA	SLA	% worse	ldrs	strs	eor
GSM	5.22	5.90	13.5	18	8	6
EPIC	1.34	2.67	99.2	228	30	24
Url	0.25	0.52	108.0	12	4	0
Md5	0.27	0.30	14.8	4	0	0
IPChains	1.69	2.67	58.0	44	14	9
Classifier	2.27	4.73	107.0	26	2	6
Firewall	1.84	2.71	47.3	24	0	6

	Exec time characteristics		
	Execution time (sec)		
Bench	w/o SLA	SLA	% imp
GSM	0.57	0.55	**3.6**
EPIC	0.65	0.61	**6.2**
Url	6.32	6.27	**0.8**
Md5	0.75	0.73	**2.7**
IPChains	0.23	0.20	**15.1**
Classifier	2.71	2.70	**0.8**
Firewall	3.49	3.41	**2.4**

	Compiling	Exec time
Bench	Overhead	Improvement
GSM	13.5	3.6
EPIC	99.2	6.2
Url	108.0	0.8
Md5	14.8	2.7
IPChains	58.0	15.1
Classifier	107.0	0.8
Firewall	47.3	2.4

bench	Transformations			%imp
	loads	stores	eor	
GSM	18	8	6	3.6
EPIC	228	30	24	6.2
Url	12	4	0	0.8
Md5	4	0	0	2.7
IPChains	44	14	9	15.1
Classifier	26	2	6	0.8
GSM	24	0	6	2.4

Conclusion

- Implemented SLA in gcc for StrongARM, studied the behavior.

- Code generated will always run faster. We found improvements in the range 0.8-15.1%.

- Performance gained depends mostly on the number of replacements in the most frequently executed code.

- As the gap between processor speed and memory latency continues to widen, such optimization will be increasingly important.

Future Work

- Merge SLA with register allocation.
- SLA for global variables.
- Use dynamic weights using profiling.
- Use heuristics and compare with our performance.

Thanks!