Trace Scheduling

**problem**: not everything is a loop

**idea**: for general non-loop situations

- find common paths in program
- realign *basic blocks* to form straight-line *trace*
  - *basic block*: single-entry, single-exit instruction sequence
  - *trace (aka superblock, hyperblock)*: fused *basic block* sequence
- schedule instructions within trace
- create fixup code outside trace in case trace != actual path
  - this can be pretty nasty
- *trace scheduling*
  - [Ellis,’85]
Trace Scheduling Example

\[ A = Y[i]; \quad \# A \text{ in } f2 \]
\[ \text{if (} A == 0 \text{)} \]
\[ \quad A = W[i]; \]
\[ \text{else} \]
\[ \quad Y[i] = 0; \]
\[ Z[i] = A*X[i]; \]
\[ \#0: \text{ldf } f2,Y(r1) \]
\[ \#1: \text{bfnez } f2,\text{instr}\#4 \]
\[ \#2: \text{ldf } f2,W(r1) \]
\[ \#3: \text{jump instr}\#5 \]
\[ \#4: \text{stf } f0,Y(r1) \]
\[ \#5: \text{ldf } f4,X(r1) \]
\[ \#6: \text{mulf } f6,f4,f2 \]
\[ \#7: \text{stf } f6,Z(r1) \]

- scheduling problem: separate \#6 (3 cycles) from \#7
  - but how to move \texttt{mulf} (and \#5 \texttt{ldf}) above if-then-else?
Basic Blocks and Superblocks

- choose most common path: A,C,D
  - assumes you know branch #1’s frequency (e.g., via profiling)
- fuse into one large “superblock” & schedule
- create repair code just in case real path was A,B,D ...

4 basic blocks: A,B,C,D
Repair Blocks

superblock (A,C,D)

#0: ldf f2,Y(r1)
#1: bfeqz f2,#2
#4: stf f0,Y(r1)
#5: ldf f4,X(r1)
#6: mulf f6,f4,f2
#7: stf f6,Z(r1)

repair code

#2: ldf f2,W(r1)
#5′: ldf f4,X(r1)
#6′: mulf f6,f4,f2
#7′: stf f6,Z(r1)

• change sense of branch condition (bfnez to bfeqz)
• repair block: may need to duplicate code (block D here)
• haven’t scheduled superblock yet ...
Superblock Scheduling 1

Superblock

\[
\begin{align*}
&\text{#0: } \text{ldf } f2, Y(r1) \\
&\text{#1: } \text{bfeqz } f2, #2 \\
&\text{#5: } \text{ldf } f4, X(r1) \\
&\text{#6: } \text{mulf } f6, f4, f2 \\
&\text{#4: } \text{stf } f0, Y(r1) \\
&\text{#7: } \text{stf } f6, Z(r1)
\end{align*}
\]

Repair Code

\[
\begin{align*}
&\text{#2: } \text{ldf } f2, W(r1) \\
&\text{#5': } \text{ldf } f4, X(r1) \\
&\text{#6': } \text{mulf } f6, f4, f2 \\
&\text{#7': } \text{stf } f6, Z(r1)
\end{align*}
\]

First scheduling move: move #5, #6 above #4

- moved load (#5) above store (#4)
- we can tell this is OK, but can the compiler?
  - if yes, fine
  - otherwise, compiler needs to do something
ISA Support for Load-Store Speculation

superblock

#0: ldf f2, Y(r1)
#1: bfeqz f2, #2
#5: ldf.a f4, X(r1)
#6: mulf f6, f4, f2
#4: stf f0, Y(r1)
#7: stf f6, Z(r1)
#8: chk.a f4, #9

repair code

#2: ldf f2, W(r1)
#5’: ldf f4, X(r1)
#6’: mulf f6, f4, f2
#7’: stf f6, Z(r1)

• change #5 to advanced load, \texttt{lf.a}
  • “advanced” means advanced past unknown store

• processor tracks load address, matches with other stores

• insert \texttt{chk.a} to check store collision. If collision? repair

• called “memory conflict buffer (MCB)”, adopted by IA64
Superblock Scheduling 2

superblock

\#0: ldf f2, Y(r1)
\#5: ldf.a f4, X(r1)
\#6: mulf f6, f4, f2
\#1: bfeqz f2, \#2
\#4: stf f0, Y(r1)
\#7: stf f6, Z(r1)
\#8: chk.a f4, \#9

repair code

\#2: ldf f2, W(r1)
\#6': mulf f6, f4, f2
\#7': stf f6, Z(r1)

second scheduling move: move \#5, (load) \#6 above \#1 (branch)
  • that’s OK, since load did not depend on branch
    • was going to execute anyway --> not speculative

scheduling non-move: don’t move \#4 (store) above \#1 (branch)
  • why? hard (but possible) to undo a store in repair block
Superblock Scheduling

what if #1 (branch) was biased the other way?
Superblock Scheduling 3

move #2 (load), #5, and #6 above #1 (branch)
  • rename f2 to f8 to avoid name conflicts
  • is this an OK thing to do?
    • from a store standpoint, yes
    • what about from a fault standpoint? what if #2 faults?

superblock

#0: ldf f2, Y(r1)
#2: ldf f8, W(r1)
#5: ldf f4, X(r1)
#6: mulf f6, f4, f8
#1: bfnez f2, #4
#7: stf f6, Z(r1)

repair code

#4: stf f0, Y(r1)
#6': mulf f6, f4, f2
#7': stf f6, Z(r1)
ISA Support for Load-Branch Speculation

• change #2 to *speculative load, ldf.s*
  • “speculative” means speculative above unknown branch
• similarly, change #6 to speculative multiply, *mulfs.s*
• processor keeps interrupt bits with registers *f8,f6*
• interrupt handled when *f6* is used by non-speculative #7
• called “poison bit” or “deferred interrupt”, adopted by IA64
Hyperblock Scheduling

what if branch #1 is not biased?

- create a large block from both paths (all 4 basic blocks)
- called a hyperblock
- use predication to conditionally execute instructions
ISA Support for Predication

Hyperblock

\[
\begin{align*}
\#0 & : \text{ldf } f2, Y(r1) \\
\#1 & : \text{sltip } p1, f2, \#0 \\
\#2 & : \text{ldf.p } f2, W(r1), p1 \\
\#4 & : \text{stf.np } f0, Y(r1), p1 \\
\#5 & : \text{ldf } f4, X(r1) \\
\#6 & : \text{mulf } f6, f4, f2 \\
\#7 & : \text{stf } f6, Z(r1)
\end{align*}
\]

- change branch #1 to \textit{set-predicate instruction, sltip}
- change instructions #2 and #4 to \textit{predicated instructions}
  - \textit{ldf.p} perform load instruction if predicate is true
  - \textit{stf.np} perform store instruction if predicate is not-true
Predication

two levels of predication

- **full predication**: can tag every instruction with predicate
  - adopted by IA64

- **conditional register moves**: (CMOVE)
  - construct appearance of full predication from one basic primitive

```c
cmoveqz r1, r2, r3          // if (r3 == 0) r1 = r2;
```

- may require a lot of code duplication
  - adopted by Alpha, IA32

- “if-conversion”: converts control-flow to data-flow
  + eliminates branches
  - why can it be bad?
Static Scheduling Summary

• loop unrolling
  + reduces branch frequency
  – expands code size, have to handle “extra” iterations

• trace scheduling
  + works for non-loops
  – more complex than unrolling and software pipelining

• ISA support
  • speculative loads, advanced loads
  • predication

This is just the tip of the iceberg - you should take a class on compilers to cover this material in more depth
Where We Stand Now

We have covered the following topics:

• performance and benchmarking
• pipelining
• dynamic scheduling
• static scheduling

next up: the memory system (caches, memory, etc.)