Adaptive and Cooperative Execution

Rodric M. Rabbah

parts of this talk are based on an ASPLOS 04 paper with Sandanagobalane, Ekpanyapong, and Wong

MIT Computer Architecture Workshop 2004
“Static” Nature of Programs

• Programs are very static and rigid
  - They do not quite adapt to runtime scenarios per se
    • Rely on out-of-order execution in some cases

• More and more programs have increasing resources available to them
  - Compiler technology is not that great
  - Proebsting’s Law: compilers double the performance of “typical” programs every 18 years
  - What to do with all that silicon?
Helper Threads

- Recent architectures with multiple execution contexts (SMT, CMP) allow processors to exploit parallelism in control-independent instruction streams

- Can we use thread contexts to "help" the program run better?
  - Speedup up single-thread workloads
  - Since 2001, lots of papers on the topic
    - ISCA, Micro, PLDI and most recently CGO
Helper Thread Example - Prefetching
Helper Thread Example - Prefetching
Helper Thread Example - Prefetching
Problems With Helper Threads

• Non-trivial overhead
  - Launching threads
  - Context switching and scheduling non-determinism
  - SMT can help but it’s not enough
  - With specialized architectural support, overhead can be reduced to ~1500 cycles

More Problems With Helper Threads

• Synchronization with the main thread
  - Is the helper thread still helping?
  - Is thread throttling and runtime adaptation possible?

• Are threads really a commodity?
  - Shouldn’t we use threads for real parallelism instead?
Better Ideas?

• IPC for many benchmarks (e.g., SPEC INT) is low
  - On VLIW architectures, it isn’t much greater than one-way parallelism
    • Itanium 2 is a 6-issue processor with 256 registers
    • With SMT (hyperthreading) used for helper threads, resource utilization is not a whole lot better (~2 ?)

• VLIW mentality: expose architecture
  - Can a compiler embed the helper thread instructions within the main (host) program?

• Why not?
  - PEPSE: program embedded precomputation via speculative execution
  - Some drawbacks as you’ll see later... but there is hope
PEPSE Overview

- Identify precomputation chain
  - For prefetching, what address to fetch from?
  - Inspect program dependence graph and identify load dependence chain (LDC)
    - Subset that computes the address

```
R1 = &list
R5 = 0
loop:
  w1:  R2 = R1 + 4
  w2:  R3 = *[R2]
  w3:  R4 = R1 + 8
  w4:  R1 = *[R4]  # problem load
  w5:  R5 = R5 + R3
  w6:  br loop (R1 != NULL)
```
PEPSE Example

• “Steal” available resources to schedule the operations in the load dependence chain

The load in $w_4$ is delinquent. Its LDC is:

$p_2$: $R1 = *[R4]$  # second LDC operation
$p_1$: $R4 = R1 + 8$  # first LDC operation

$R1 = &\text{list}$
$R5 = 0$

loop:

$w_1$: $R2 = R1 + 4$
$w_2$: $R3 = *[R2]$
$w_3$: $R4 = R1 + 8$
$w_4$: $R1 = *[R4]$
$w_5$: $R5 = R5 + R3$
$w_6$: br loop ($R1 != \text{NULL}$)
PEPSE Example

• “Steal” available resources to schedule the operations in the load dependence chain

The load in $w_4$ is delinquent. Its LDC is:

\[
p_2: \quad R1 = *[R4] \quad \text{# second LDC operation}
p_1: \quad R4 = R1 + 8 \quad \text{# first LDC operation}
\]

\[
R1 = &\text{list} \\
R5 = 0 \\
R6 = R1 + 8
\]

\[
\text{loop:}
\]

\[
w_1: \quad R2 = R1 + 4; \\
w_2: \quad R3 = *[R2] \\
w_3: \quad R4 = R1 + 8 \\
w_4: \quad R1 = *[R4] \\
w_5: \quad R5 = R5 + R3; \\
w_6: \quad \text{br loop (R1 != NULL)}
\]

\[
p_2: \quad R7 = *[R6] \\
p_1: \quad R6 = R1 + 8
\]
Preliminary Itanium 2 Results

• Implemented prototype algorithm in ORC for Itanium Processor Family
  - Open-source parallelizing compiler
  - Used scientific benchmark set
    • SPEC FP (SPEC INT results later)
    • Lots of available resources to exploit
  - Compared results against
    • Built-in prefetching (Mowry's thesis)
    • Software pipelining

• Compared to the best ORC baseline, PEPSE reduces total runtime of 9 benchmarks by 13 minutes (27%)
PEPSE vs. Helper Threads

• Most significant difference: program counters
  - Helper threads have a dedicated PC
  - PEPSE is part of the main program instruction stream and shares the PC

• When is this a problem?

```
... LD r1 = [r0]  # cache miss
... ADD r1 = r1, 4  # processor stalls
LD r2 = [r1]...
```

• As long precomputation is on-path, visible effect is shifting stalls to occur earlier in time
What Now?

• Precomputation must adapt
  - Abandon when it appears not profitable

    ...  
    LD   r1 = [r0]  # cache miss  
    ...  
    ADD r1 = r1, 4  # want to ignore this operation  
    LD   r2 = [r1]  # and this one  
    ...

• Use predication

• SPEAR: sentineled precomputation for EPIC architectures
SPEAR Example

• Precomputation must adapt
  - Abandon when it appears not profitable

...  
ILD  r1 = [r0]  # informing load, on cache miss, p ← 1  
...  
ADD r1 = r1, 4  if ¬p  # conditionally issue this operation  
LD  r2 = [r1]  if ¬p  # and this one  
...

• Program adapts to runtime behavior
• Architecture cooperates with the program to provide important runtime information
  - Must cheaper than threads, many ISA tricks apply
Preliminary Results

• Implemented prototype system using Trimaran
  - Open source compiler for VLIW research
  - Cycle accurate simulator configured to resemble an Itanium processor
  - Used SPEC INT and SPEC FP benchmarks
    • As might be expected, SPEAR has little advantage for array codes
    • In integer (pointer heavy) codes, 13% additional improvement over PEPSE
    • Can reduce processor stalls 45% on average and 70% or more in the best cases
Room To Improve

- Overlapping precomputation chains
  - Reinforcement in data dependent precomputation

- Hybrid chains
  - Investment vs. Payoff in precomputation chain

```
LD r1 = [r0]  # cache miss, costs 7 cycles
...
ADD r1 = r1, 4  # wait for result
LD r2 = [r1]  # issue prefetch, save 5 cycles 😞
...
```
Room To Improve

- Overlapping precomputation chains
  - Reinforcement in data dependent precomputation

- Hybrid chains
  - Investment vs. Payoff in precomputation chain

... 
LD r1 = [r0]  # cache miss, wait 5 cycles
...
ADD r1 = r1, 4 # wait for result
LD r2 = [r1] # issue prefetch, save 7 cycles 😊
Other ACE Applications

• Branch prediction
  - Tackle data-dependent branches
  - Hopeless for deep pipelines
    • Have to precompute results 30+ cycles in advance
    • Branch condition usually on the critical path
  - But processors are changing
    • Itanium 2 has an 8-stage pipeline
    • There is hope!

• Other examples
  - Address disambiguation
  - Voltage scaling
  - Resource allocation (wait for Dave’s talk)
Thanks!