Some Cache Optimization with Enhanced Pipeline Scheduling

Seok-Young Lee, Jaemok Lee, Soo-Mook Moon

School of Electrical Engineering & Computer Science

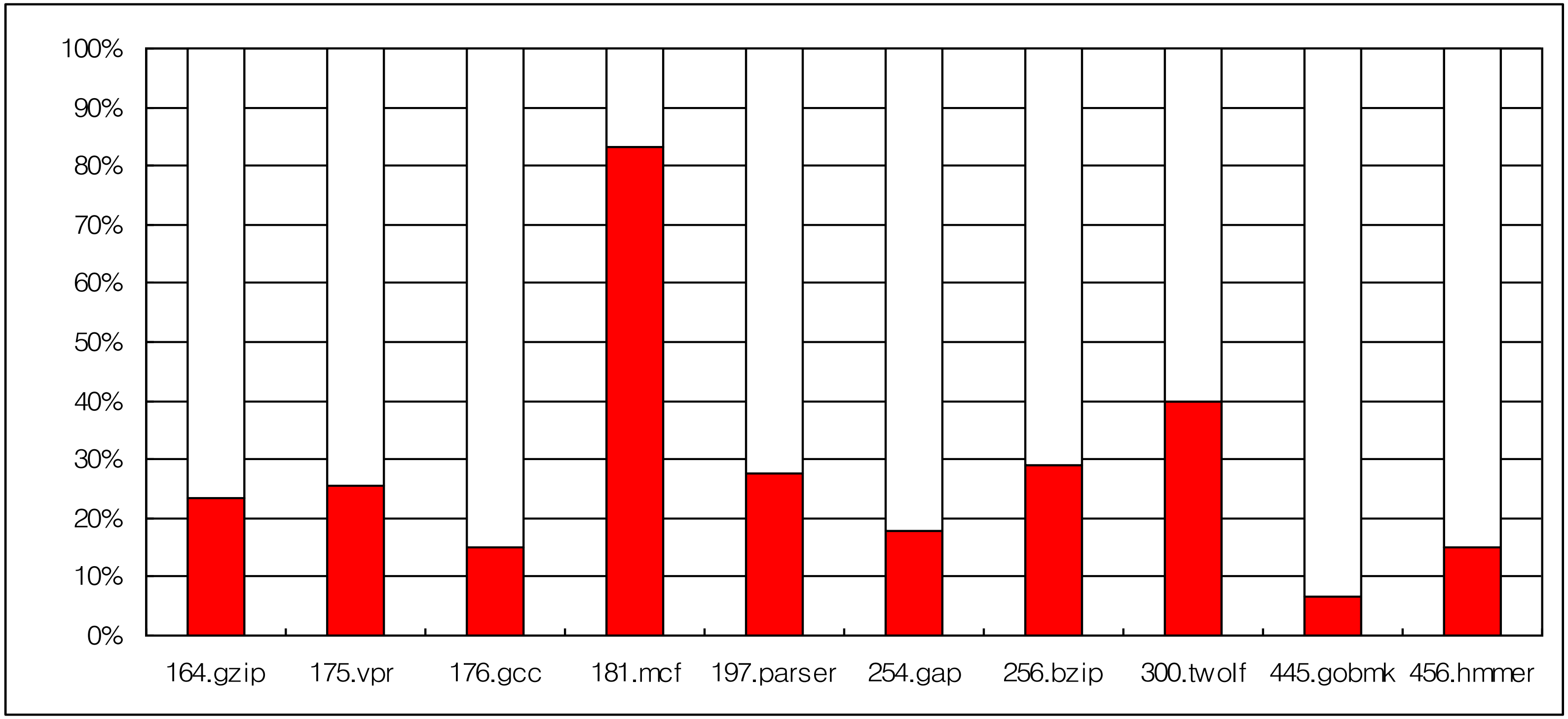
Seoul National University, Korea

Outline

- Motivation and background
- Cache optimizations with Enhanced pipeline scheduling
- Experimental results
- Summary and future work

Cache Misses for Integer Programs

- CPU stalls caused by data cache misses are serious, even in some integer programs



CPU Stall portion in the total running time

Conventional Techniques

- Many compiler optimization techniques have been used
  - Prefetches for array-accessing loops [Mowry'92]
  - Increasing locality in loops [Wolf'91]
  - Dynamic runtime optimization [Chilimbi'02]
- But they are not well applicable to integer loops
  - Address estimation is not easy (e.g., pointer-chasing loops)
  - Complex control flows

A Better Technique

- In integer programs, it is easier to separate "hot cache-missing loads" from their consumers by cache-miss latencies
  - Simply implemented by increased load latency during code scheduling

load **x** = [y]

use **x**

CPU stall if cache miss

use a
use b
use c
use d
load **x** = [y]

Cache miss latency

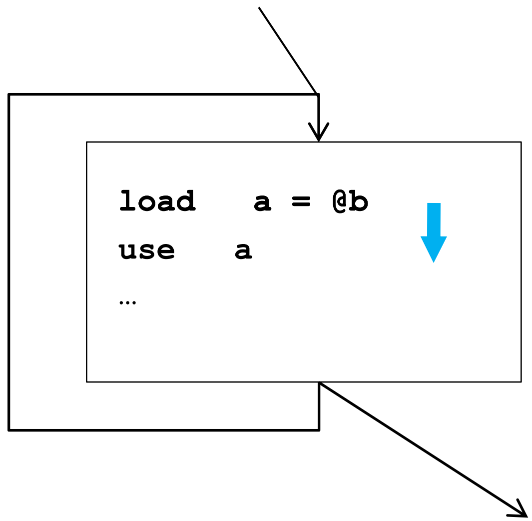
use **x**

No CPU stall
if the load and consumer
Is separated.

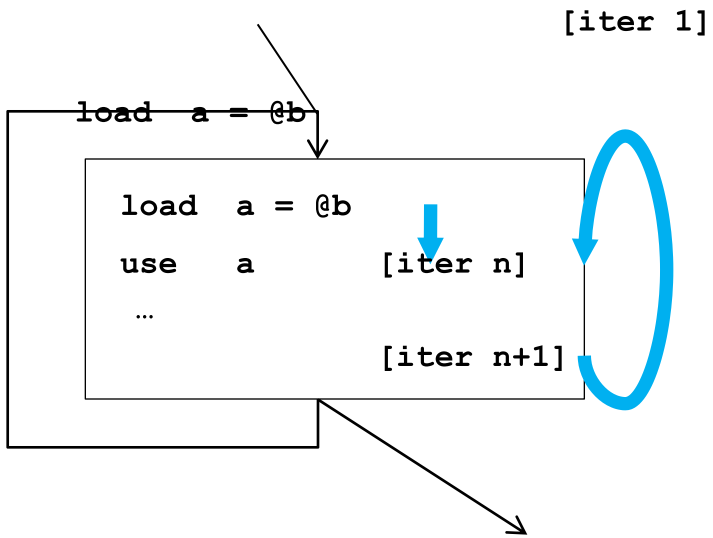
Our Proposal

- However, naïve code scheduling is not enough
  - Code motion of hot loads can be stuck at the loop entry
  - Difficult to fill added slack cycles fully and usefully
  - Actually, did not show tangible impact [Choi ’02 in EPIC-2]
- Our proposal: moving hot loads across loop iterations

Illustration of the Proposal



naïve separation:
stuck at the loop header



proposed separation :
moving hot loads across loop iterations
A code motion for software pipelining

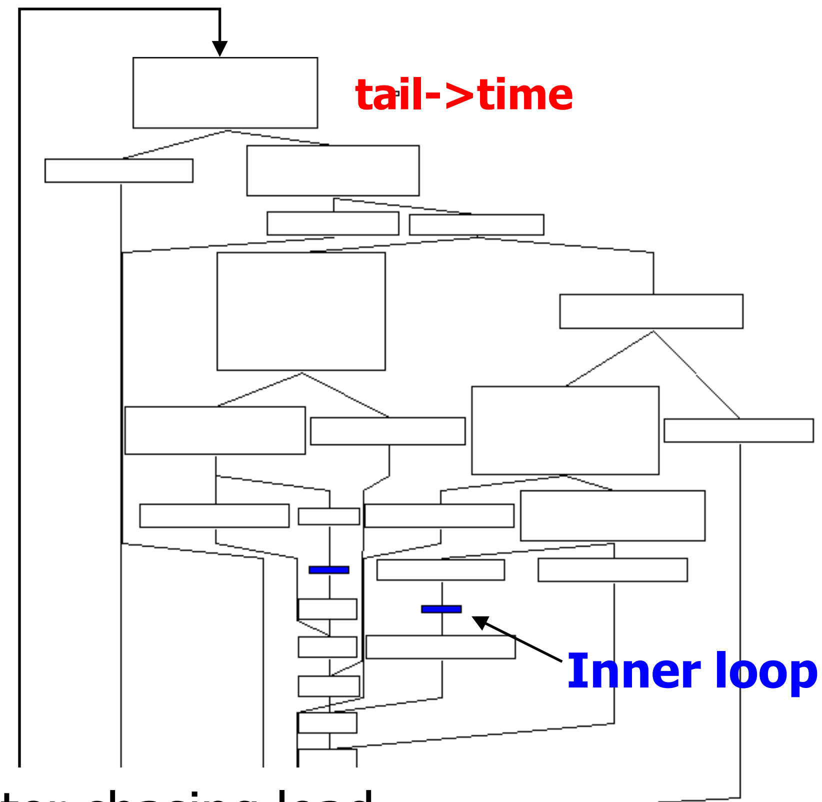
Some Characteristics of Hot Loads

- Located close to loop entry
- Tight data dependence chains to their source operands
  - Moving hot load requires moving dependent instructions as well
- Difficult to estimate target address
- Often in a loop with complex control flow
  - Require code motion above branches and joins

Hot load example in 181.mcf

```
while( arcin )
{
  tail = arcin->tail;
  if( tail->time + arcin->org_cost > latest )
  {
    arcin = (arc_t *)tail->mark;
    continue;
  }
 …
}
```

Complex and large code including inner loop and function call



Pointer chasing load

In an outer loop with complex control flow

Close to the loop entry

181.mcf source code

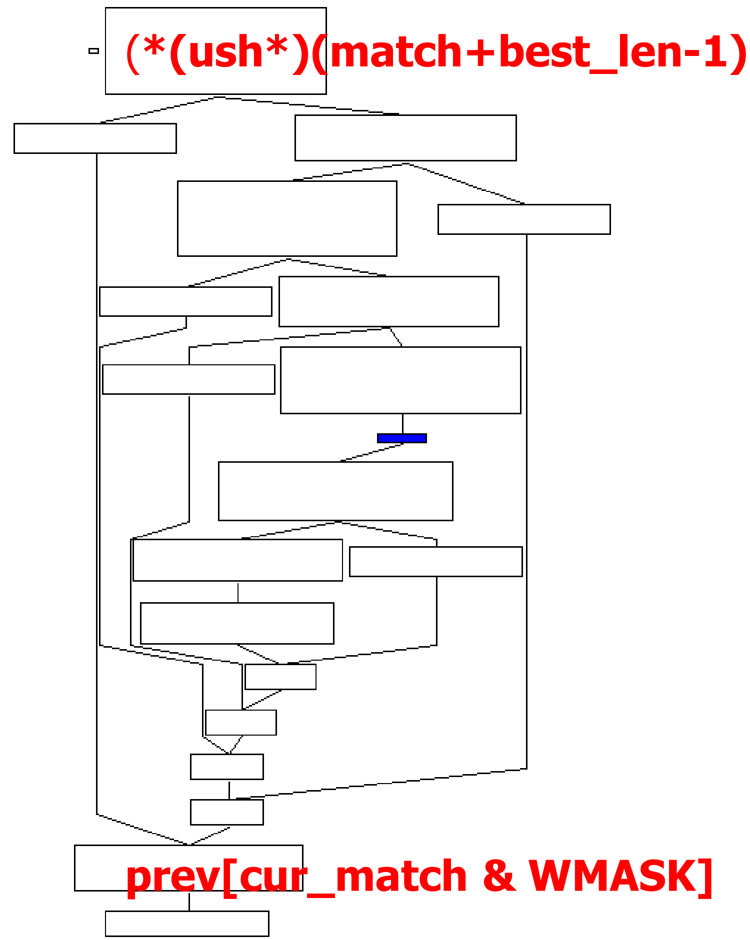
181.mcf control flow graph

Hot load example in 164.gzip

```
do {
  match = window + cur_match;
  if (*(ush*)(match+best_len-1) != scan_end ||
      *(ush*)match != scan_start) continue;
```

Complex and large code including inner loop and function call

```
} while ((cur_match = prev[cur_match & WMASK])
            > limit && --chain_length != 0);
```

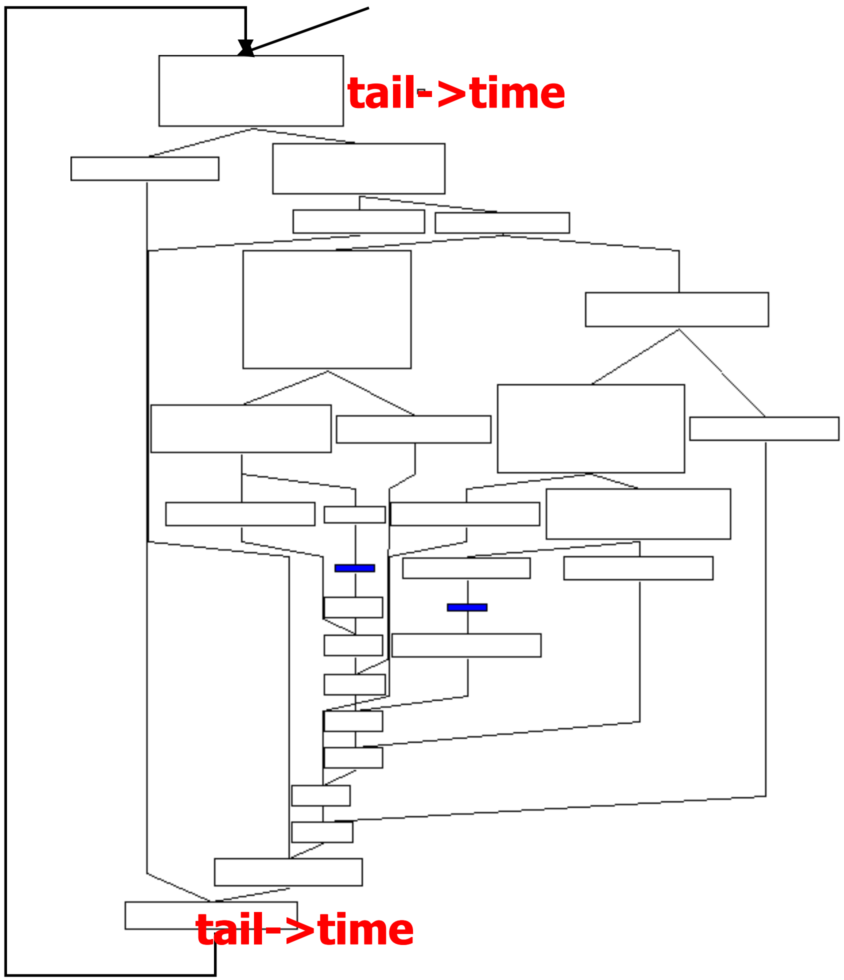


164.gzip source

164.gzip control flow graph

Cross-Iteration Global Scheduling

- Separating hot loads requires two types of code motions
  - Code motion across loop back-edges: software pipelining
  - Code motion across branches and joins: global scheduling



- Needs global scheduling across loop iterations

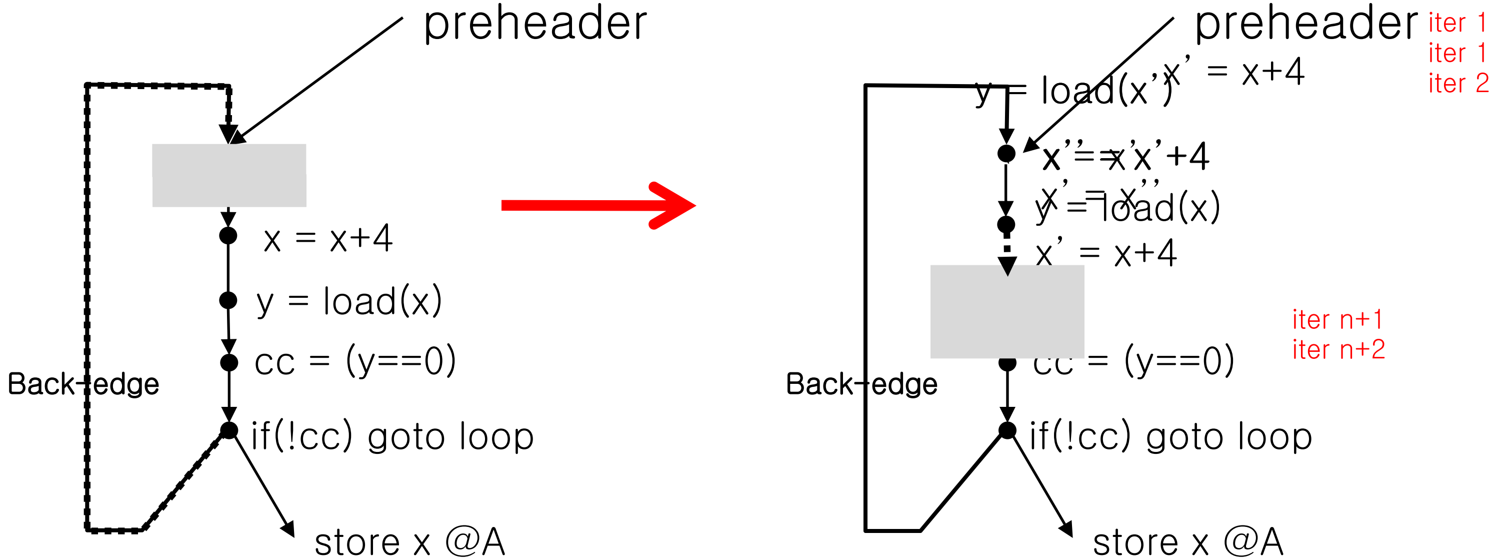
➔ Enhanced pipeline scheduling

Enhanced Pipeline Scheduling (EPS)

- A software pipelining technique based on code motions
  - Global scheduling can be applied across loop back-edges
  - Aggressive code motions for scheduling useful instructions
- If we exploit EPS appropriately, we can (1) separate hot loads and the consumers effectively and (2) fill the slack cycles usefully
- Let us first review how EPS works

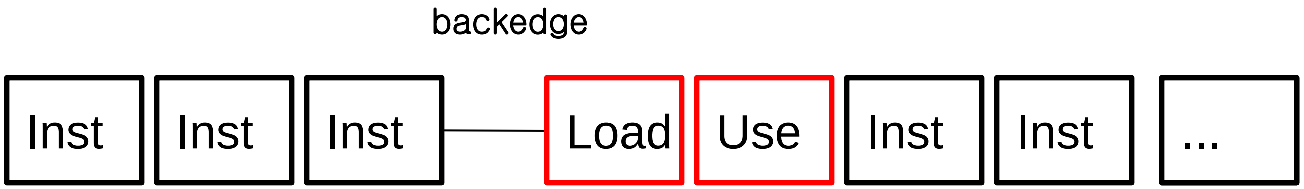
EPS Illustration

- EPS repetitively (1) defines a DAG by cutting edges of a loop and (2) performs DAG scheduling



CPU Stall Reduction with EPS

- We simply add a L1-cache-missing latency for "hot" loads and schedule them by EPS algorithm
  - Their consumer instructions will be scheduled far enough from them, even across loop iterations



- However, this is not that simple

Issues in Stall Reduction with EPS

- Adding slack cycles means more aggressive code motions
  - Some aggressive code motions such as speculative loads or join code motions have a negative side-effect if performed recklessly
  - Must limit aggressive code motion

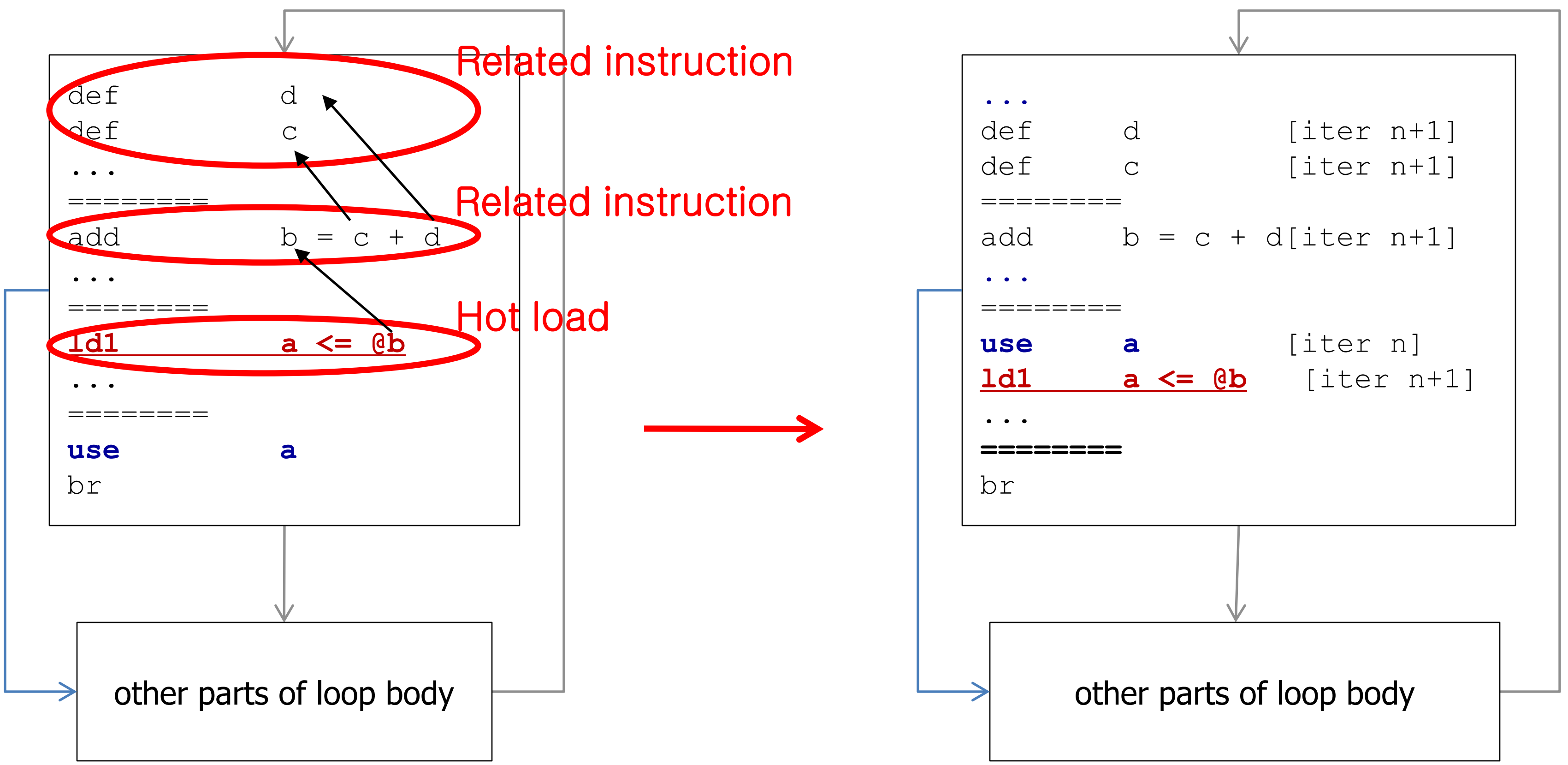
- On the other hand, hot loads and their source definitions should be scheduled aggressively
  - Must encourage aggressive code motion

Hot Load-related instructions

- We split instructions into two groups, hot-load-related instructions and non-related instructions.

- Hot-load-related instructions are scheduled more aggressively than non-related instructions
  - Selective heuristics

Scheduling Hot Load-related instructions



Stall-Reducing EPS for Open-64

- We implemented EPS into Open-64 (version 3.0), an open-source compiler for IA-64
  - http://www.open64.net/
  - EPS is positioned between register allocation and global instruction scheduling in Open-64

- We then implemented stall reduction for EPS
  - Detect “hot” loads via profiling

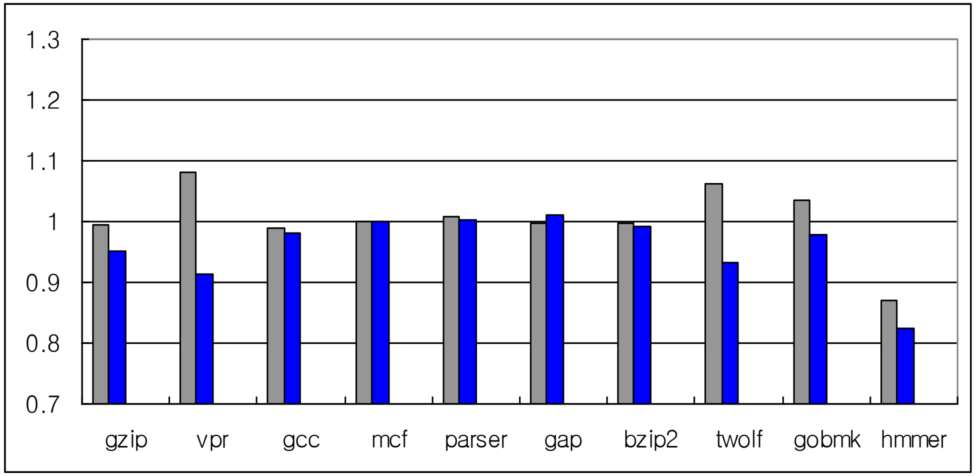
Experimental Results

- Experimental Environment
  - Intel Itanium2 processor 900Mhz
    - 256Kb L1 D-cache (L1 cache miss takes 5 Cycles)
  - 10 integer benchmarks from SPEC CPU 2000 and 2006
  - Use Performance Monitoring Unit for detecting hot loads
    - Collect load instructions whose stall overhead takes over 2% of running time
    - 12 loops in 10 benchmarks are selected
    - We do not touch other loops

Experiment Configurations

- Base: Open-64 –O3 with EPS disabled (1.0x)
- EPS without cache optimizations
  - Strictly schedule hot loops only
- EPS with cache optimizations
  - Strict heuristics
    - Limited code motions
  - Aggressive heuristics
  - Selective heuristics for hot-load-related instructions

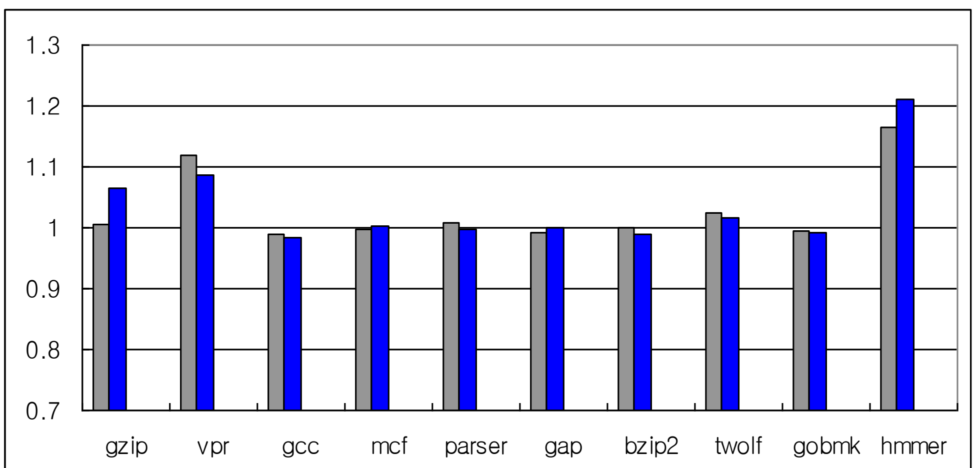
Stall Reduction and Performance Result



Stall cycles

Strict EPS without Cache Optimization

Strict EPS with Cache Optimization

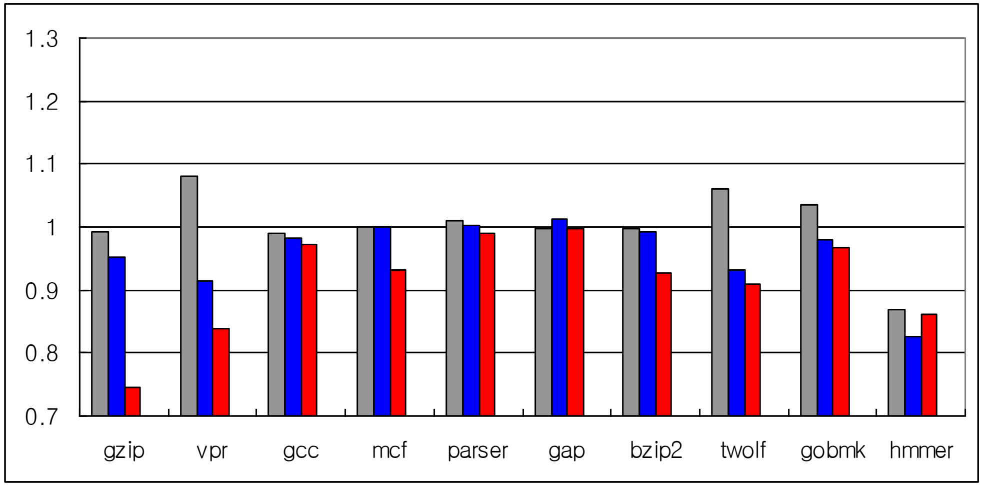


Total execution cycles

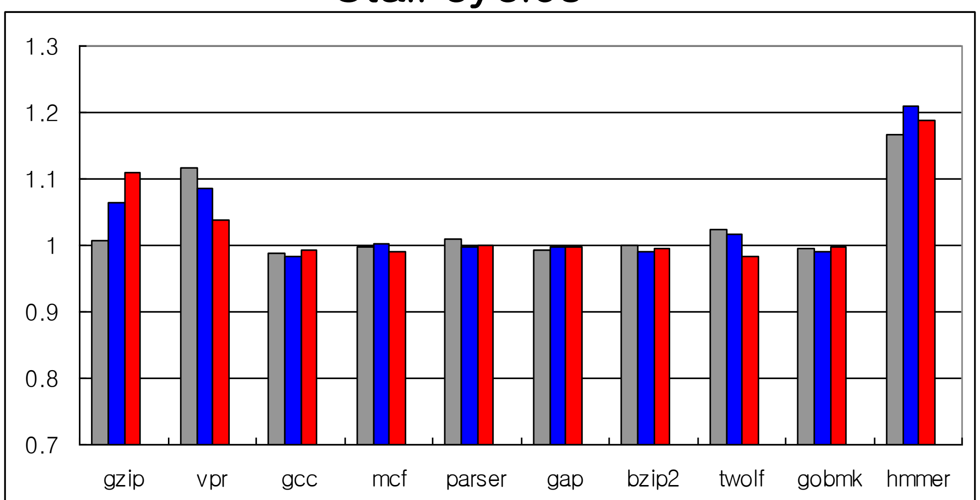
Stall is reduced a little than EPS w/o cache optimization configuration.

No tangible effects in execution cycles.

Stall Reduction and Performance Result



Stall cycles



Total execution cycles

Strict EPS without Cache Optimization

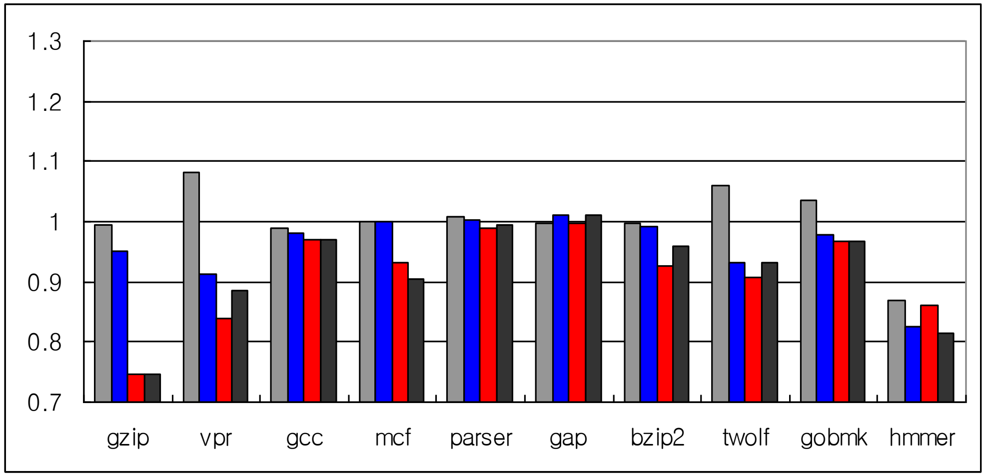
Strict EPS with Cache Optimization

Aggressive EPS with Cache Optimization

Stall is reduced more.

Execution cycle does not get better

Stall Reduction and Performance Result



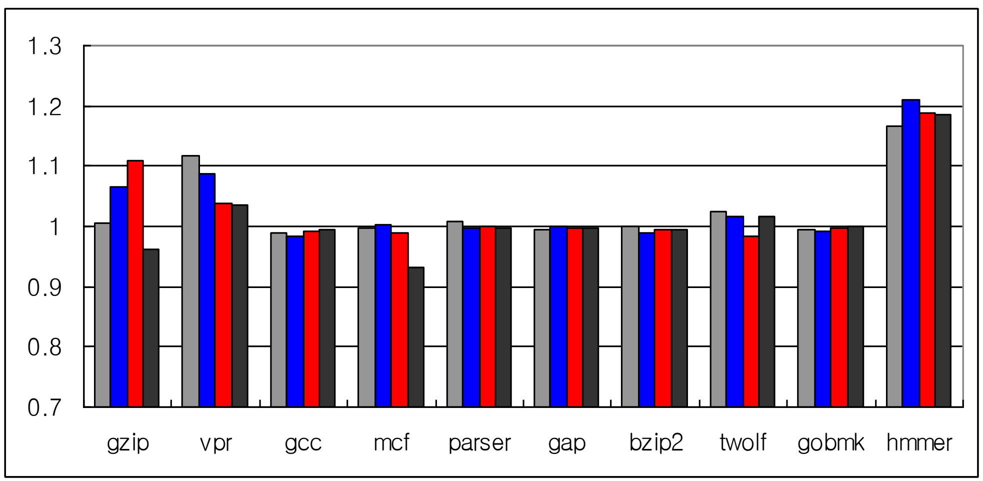
Stall cycles

Strict EPS without Cache Optimization

Strict EPS with Cache Optimization

Aggressive EPS with Cache Optimization

Selective EPS with Cache Optimization



Total execution cycles

Stall is reduced as much as aggressive configuration.

Execution cycle is decreased. Especially gzip and mcf.

Summary and Future Work

- EPS-based stall reduction achieves promising result
  - Adding L1-cache-miss latency for hot loads to separate them from their consumers
  - Aggressively schedule hot-load-related instructions only
- Future Work
  - More balanced heuristics between parallelism & stall reduction
  - Handling L2-cache-miss for some hottest loads

Thanks

- Questions?