Program Phase Directed Dynamic Cache Way Reconfiguration for Power Efficiency

Subhasis Banerjee\textsuperscript{1}    G. Surendra\textsuperscript{2}    S. K. Nandy\textsuperscript{2}

\textsuperscript{1} Sun Microsystems
Bangalore - INDIA

\textsuperscript{2} CAD Lab, SERC
Indian Institute of Science
Bangalore - INDIA
Outline

1. Exploiting Runtime Program Behavior
   - Motivation
   - Program Phases

2. Program Phase Directed Cache Reconfiguration
   - Hardware Phase Detector
   - Way Selection using Phase Information
Outline

1. Exploiting Runtime Program Behavior
   - Motivation
   - Program Phases

2. Program Phase Directed Cache Reconfiguration
   - Hardware Phase Detector
   - Way Selection using Phase Information
Power density in high performance microprocessor has doubled in every three years [Borkar(Jul-Aug 1999)]

Conventional circuit level power management techniques do not address issues related to dynamic power

Need to address power issues at program runtime

Runtime behavior studied most effectively in architecture level

Microarchitecture level power estimation is faster and the accuracy is 90% compared with circuit level technique

Solution: Power Aware and Application Aware Micro-architecture
Advantages:
- Ability to estimate power consumption at early stage of design
- Important to study performance-power trade-off quickly
- Explore opportunities to save power via micro-architecture optimization – clock gating, dynamic adaptation
- Visibility to program runtime: program behavior ← adaptive micro-architecture

Disadvantages:
- No direct solution to static power dissipation
- Accuracy is always an issue
**Aim of the Study**

- Architecture level exploration of program behavior in terms of program phase
- Characterization of program phases using cache conflict miss
- Reconfigurable cache architecture using program phase information
- Incorporate demand driven policy to allocate cache resources at program runtime
Exploiting Runtime Program Behavior

Motivation

Program Phases

Program Phase Directed Cache Reconfiguration

Hardware Phase Detector

Way Selection using Phase Information
**Program Behavior**

- **Definition:** Program phase is the observable phenomena that reflects program behavior.
- Microarchitecture optimizations, in general, depend on program behavior.
- Well known program behavior:
  - temporal and spatial locality
  - value locality
  - program predictability
- Caches and branch predictors rely on program behavior for performance improvement.
IPC, L1 data cache miss and normalized power averaged over interval of 5 million instructions are shown for gcc.

It is observed that all the metric show a repetitive phased behavior.
Program Behavior – Observation

- It is sufficient to observe program behavior at a granularity of million instructions [Duesterwald et al.(2003)Duesterwald, Cascaval, and Dwarkadas]
- Although different program metrics change differently with time, the periodicity in the behavior is the same
- Once program enters in a particular “program phase”, it remains in that phase for some time $\Rightarrow$ program locality
- It is observed that the majority of programs are control dependent i.e., the program behavior depends on the control transfer behavior of dynamic instruction stream [Sazeides and Smith(1998)]
### Simulation Environment

<table>
<thead>
<tr>
<th>Parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch queue size</td>
<td>8 instructions</td>
</tr>
<tr>
<td>RUU Size</td>
<td>64 instructions</td>
</tr>
<tr>
<td>LSQ Size</td>
<td>32 instructions</td>
</tr>
<tr>
<td>Fetch width</td>
<td>4 instructions/cycle</td>
</tr>
<tr>
<td>Decode width</td>
<td>4 instr/cycle</td>
</tr>
<tr>
<td>Issue width</td>
<td>4 instr/cycle</td>
</tr>
<tr>
<td>Commit width</td>
<td>4 instr/cycle (in-order)</td>
</tr>
<tr>
<td>Functional Units</td>
<td>4 int ALU</td>
</tr>
<tr>
<td></td>
<td>1 int mult/div</td>
</tr>
<tr>
<td></td>
<td>2 mem ports</td>
</tr>
<tr>
<td>Branch Predictor</td>
<td>Bimodal 2K table</td>
</tr>
<tr>
<td></td>
<td>2-Level – (gshare) 1K table, 10 bit</td>
</tr>
<tr>
<td></td>
<td>combined – bimodal and gshare</td>
</tr>
<tr>
<td></td>
<td>history – 1K chooser, 4 cycles penalty</td>
</tr>
<tr>
<td>BTB</td>
<td>2048 entry, 4-way</td>
</tr>
<tr>
<td>Return address stack</td>
<td>16-entry</td>
</tr>
<tr>
<td>L1 data cache</td>
<td>64k 4-way, LRU</td>
</tr>
<tr>
<td></td>
<td>32B block, 1 cycle latency</td>
</tr>
<tr>
<td>L1 instruction cache</td>
<td>64K, 2-way, LRU</td>
</tr>
<tr>
<td></td>
<td>32B block, 1 cycle latency</td>
</tr>
<tr>
<td>L2 cache</td>
<td>Unified, 1M, 4-way, LRU</td>
</tr>
<tr>
<td></td>
<td>32B block, 15 cycle latency</td>
</tr>
<tr>
<td>Memory</td>
<td>100 cycle first chunk latency</td>
</tr>
<tr>
<td></td>
<td>2 cycles subsequently</td>
</tr>
<tr>
<td>TLB</td>
<td>128 entry itlb, 128 entry dtlb,</td>
</tr>
<tr>
<td></td>
<td>4-way, 30 cycle miss latency</td>
</tr>
</tbody>
</table>

- **Watcht** simulator build on SimpleScalar – cycle accurate superscalar simulator
- Incorporated modules from HotLeakage – to estimate leakage current
- 
  - 0.13\(\mu\)m process technology parameters with 1.7 GHz clock frequency
- **Benchmarks → SPEC-2000**
  - CPU benchmarks with reduced MinneSPEC input and MEDIABENCH
Outline

1. Exploiting Runtime Program Behavior
   - Motivation
   - Program Phases

2. Program Phase Directed Cache Reconfiguration
   - Hardware Phase Detector
   - Way Selection using Phase Information

S Banerjee et. al.
A phase detector based on basic block count at runtime is proposed in [Sherwood et al.(2003)Sherwood, Sair, and Calder]

**Disadvantage** The basic blocks are randomly chosen by the hash function, inconsistent phase information in different run

Lossy technique. Aliasing effect from different basic block
Phase Detector

- Our phase detector uses cache conflict miss information to form a vector.
- Each cache set contains one tag-counter pair. Tag stores the last 8 bit of most recently evicted tag.
- The conflict miss is recorded if the tag of recent miss matches with the tag of the corresponding set.
- The interval during which the Miss Vector is accumulated is 2 million instructions.
Phase Detection Algorithm

get mc_vector[m]; {mc_vector[m] is a vector obtained in an interval m}
min = BIGNUM;
for all i = 1 to m – 1 do
    dist = distance(mc_vector[i], mc_vector[m]);
    if dist ≤ min then
        min = dist;
        index = i;
    end if
end for
if min ≤ threshold then
    update(mc_vector[index], mc_vector[m]);
else
    add_new_mct_entry(mc_vector[m]);
end if

threshold = 1.1

- Vectors are normalized
- 4 most significant bits are stored for each element of the vector
- For a cache having S sets, we need to store 4S bits per phase. 8 phases are stored in the table. The size of phase history table (PHT) is 4S bytes.
- for 512 sets, the size of the PHT = 4 × 512 = 2kB
Number of Clusters with Different Threshold

![Graph showing the number of clusters with different thresholds for various applications.](image)

- Applications: gzip, bzip2, crafty, gcc, mcf, parser, perlbench, twolf, vpr, vortex.
- Applications: ammp, applu, art, equake, facerec, galgel, mesa, mgrid, swim.
- Applications: mpeg2enc, mpeg2dec, cjpeg, djpeg, lame, caudio, daudio.
Outline

1. Exploiting Runtime Program Behavior
   - Motivation
   - Program Phases

2. Program Phase Directed Cache Reconfiguration
   - Hardware Phase Detector
   - Way Selection using Phase Information
Cache Reconfiguration

Basic Cache Architecture

S Banerjee et. al.
Wordline and Bitline Segmentation: Optimal $N$ Parameter

- $N_{dwl} \Rightarrow \# \text{ of wordline segments} \geq \text{associativity}$
- Each way therefore spans into a number of wordline segment
Cache controller keeps track of the program phases detected by the phase detector

Cache ways are enabled/disabled (if needed) using a way selection algorithm at the end of every interval of 2 Million instructions

Reconfigurable cache organization
Cache Way Selection Algorithm

```python
if (STATE == STABLE) then
    if ((present_miss - recorded_miss) < miss_noise) then
        miss_noise = noise_dcr;
        update_state();
    else
        shutdown_one_way_of_Cache;
        update_state();
        STATE = UNSTABLE;
    end if
else
end if

if (STATE == UNSTABLE) then
    if (miss_rate > threshold) and (available_ways != 0) then
        enable_one_more_way;
        update_state();
    else
        state = STABLE;
        update_state();
        miss_noise = base_noise;
    end if
end if
```

- The algorithm finds a suitable configuration at the end of every interval.
- Every new phase starts with UNSTABLE state.
- If miss_rate is within limit, STABLE state is assigned.
- To avoid frequent configuration, a base_noise is set.

<table>
<thead>
<tr>
<th></th>
<th>SPEC INT/FP</th>
<th>MEDIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>threshold</td>
<td>3%</td>
<td>2%</td>
</tr>
<tr>
<td>base_noise</td>
<td>4000</td>
<td>3000</td>
</tr>
<tr>
<td>noise_dcr</td>
<td>200</td>
<td>100</td>
</tr>
</tbody>
</table>
Effect of Disabling Cache Ways

- Data consistency should be maintained while valid data are residing in disabled way

- Three methods are studied:
  - All data in the disabled way are flushed ⇒ simplest but significant performance loss
  - A fill buffer approach to move data from disabled to enabled way. Modification is required in the datapath to move data from disabled to enabled way
  - An accessed block in a disabled way is stored in a 4 entry fully associative victim buffer
Fill Buffer Approach and Victim Buffer for Data Consistency

- Fill buffer approach: Data transferred to enabled way takes 8 cycles
- A request for a data in disabled way is moved to the victim buffer
- Eviction from the victim buffer writes data back to L2 cache
- Victim buffer has a 4 entry fully associative structure, more energy dissipation

Figure: (i) Fill buffer approach, (ii) Victim buffer
We incorporate victim buffer approach

More than 75% of the accesses in the disabled ways are serviced by the victim buffer

It avoids frequent activation of precharge and sense amplifier logic of the disabled way

Every hit to victim buffer saves 7 cycles penalty compared to fill buffer approach
Hardware Overhead

- Hardware consisting of a tag associated with saturating counter, number of such tag-counter pair is equal to the number of sets
- 8-bit tag part of the phase detector contributes the major part in power
- The tag part is modeled as a direct-mapped cache tag which is indexed by the index bits of the address
- Our estimation shows that the power overhead averaged over all benchmarks is around 2% of the power dissipated by the cache
- The power consumed by victim buffer is modeled using the SRAM power model used in Wattch simulator
Performance with Cache Way Reconfiguration

Figure: IPC and EDP while compared with the **base case – 4 way assoc**
Result Summary

- Result is reported with all combination of base case configuration
- Direct mapped cache is obviously the least in power budget, IPC performance is worst
- 2-way set associative cache shows too much variation in IPC and energy dissipation
- Gain over 4-way associative shows that average energy saving in L-1 data cache is 28-32% while the performance (IPC) loss is only 1-2%
- The size of additional hardware is approximately 5% of the size of the cache
Conclusions

- High associativity in cache improves performance, but a static allocation of all the cache ways is not energy efficient.
- Cache utilization metric is derived using conflict miss information from all cache sets.
- Runtime behavior of program can be effectively captured by the hardware phase detector to estimate cache utilization.
- Way partitioned caches can be configured successfully at runtime for energy efficiency.
References

S Borkar.
Design challenges of technology scaling.
4

E. Duesterwald, C. Cascaval, and S. Dwarkadas.
Characterizing and predicting program behavior and its variability.
10

Y Sazeides and J E Smith.
Modeling program predictability.
10

Timothy Sherwood, Suleyman Sair, and Brad Calder.
Phase tracking and prediction.
13
THANK YOU

QUESTIONS ?