Profile-driven Intelligent Shared-Cache Allocation

Weidan Wu
ECE Dept., Duke Univ.
ww53@duke.edu

Yunjia Zhou
CS Dept., Duke Univ.
yunjia@cs.duke.edu

Fan Yang
CS Dept., Duke Univ.
fanyang@cs.duke.edu

Abstract

Application behavior is powerful hints to dynamic optimization, since the variance of intra- and inter- applications poses distinct requirements on resource management, thus leading to different optimization policy making. Although it is feasible to profile applications in hardware or in software alone, they suffer from limitation of implementation overhead or monitoring coverage respectively.

This paper demonstrates a cross layer manager which combines the advantages of OS and CPU in shared cache management, which dynamically captures the variant behavior of applications. The results show that, using simple LRU stack simulation, we are able to accurately capture more than 95% of cache hit/miss behavior; while the overhead, which is majorly the exception interface between CPU and OS, is reasonably low. This work provides a framework of profile driven optimization which leverages the advantage of both OS and CPU in dynamic optimization; it enables interesting hardware software co-design which aims to maximize performance while minimize cost.

1. Problem Motivation

Multiple programs execution incurs interferences which, if not properly handled, may reduce performance due to shared resource contention and load imbalance. Thus, approaches are necessary to govern the shared resource allocation [5] and monitor the asymmetrical behavior of processes, where the system manager requires detailed run time information and complex computation to develop an optimal resource allocation. On the other hand, active low power states [4] are an efficient power saving techniques. In power constraint computing, allowing part of the resource in active low power states is an appealing mechanism to minimize power consumption without significantly impact performance. However, it is not feasible to depend only on either the operating system or the hardware [1], respectively. The operating system lacks the low level details of run time information, such as cache misses, while the hardware can hardly perform such complex computation due to expensive hardware resources and inflexible hardware implementation. Therefore, a cross layer effort is needed combining the operating system and the hardware to achieve desirable performance.

2. Design Goals

In this paper, a cross-layer mechanism is designed for more intelligent runtime management. A feedback loop is built up between OS and CPU, where CPU profiles memory access stream while OS makes bandwidth optimization policy based on the profiled information. An on-line miss rate curve (MRC) is developed using stack simulator. The CPU utilizes built-in hardware monitors to profile the memory access sequences and provides them to the stack simulator in OS for developing MRC on line. Based on MRC, an OS manager is able to infer the cache requirement of each program and
perform cache allocation according to their individual requirements.

Software behavior exhibits phase patterns. As a result, hardware monitors should also be able to detect phase change based on other characteristics, and the runtime manager can make different decisions in different situations. In this work, we design an interval-based approach, which temporally stores the memory access profile in hardware and periodically sends to the OS for MRC curve fitting. More complex mechanisms are possible, like buffer overflow triggering, or application phrase change triggering. But all of these rely on the same fundamental mechanism demonstrated in this work.

We believe that this hardware-software co-design leads to more intelligent cache allocation. More generally, it provides a framework of observation driven optimization on heterogeneous platforms.

3. Design Architecture

As mentioned in last section, our design goal is to build a cross layer manager to achieve OS-CPU co-operation. Our work implements a feedback loop between OS and CPU.

We divide the whole process into 3 steps:

1) CPU profiles memory access stream;
2) OS makes optimized cache allocation policy based on the profiled information;
3) CPU makes the change accordingly.

More specifically, we develop an on-line miss rate curve (MRC) using stack simulator similar to previous work. The CPU uses built-in hardware monitors to profile the memory access sequences and provides them to the stack simulator in OS for developing MRC on line. Based on MRC, an OS manager is able to infer the memory requirement of each program and perform cache allocation according to their individual requirements.

3.1. Hardware Profiling

A significant part of our design is hardware profiling which provides the detailed runtime information of applications. In this phrase, the simulator collects memory access, and is ready to send it to OS for further processing. The memory access stream originally contained per memory address sequence, which was not necessary in building MRC, and was too expensive in both collecting, transferring and computing. We reduced the overhead by collecting per block accesses; in other words, we masked out the in-block address bits and only send the block addresses to OS. Because cache miss granularity is per block, this handling still maintains the original cache miss pattern. Furthermore, we aims at studying shared cache which, in our current configuration, is Level 2 cache. Thus we only collect Level 1 cache misses, which is roughly 2% of the total memory accesses. These two factors make the hardware overhead acceptable to be part of the shared Level 2 cache or main memory.

A further optimization is possible if we collect the memory access stream based on multiple block sizes, which is 32 bytes, 64 bytes, 128 bytes, 4096 bytes and 8192 bytes. On one hand, this allows more versatile manager which deals with different cache implementations. While the program only needs to run once, the decision covers multiple situations. On the other hand, memory accesses pattern over page granularity combines with OS page fault, allow MRC creation on memory level, rather than limited in cache level, thus allowing broader coverage of our manager. It should be noted that this multiple block method does not incur much extra overhead because different logic stacks can share the same physical stack.
3.2. Hardware-software communication

After collecting memory access stream from hardware, we will send these data to OS for MRC curve development via exception. The hardware generates an exception for every 10 million instructions interval. A dedicated exception is defined in the CPU and the corresponding handler entry is registered in the Linux kernel to handle this hardware software communication. At the same time, an unused memory region of Alpha memory space is used to allow data returned to the OS, which essentially move data from the dedicated hardware buffer to the virtual memory space of the OS kernel.

A possible alternative is to design special instruction to manipulate the dedicated hardware buffer. This allows no change in the memory unit, which is always in the critical path of the pipeline. However, this design requires extra decoder enhancement; moreover, it changes the ISA which defines the functionality of the architecture. On the other hand, our current design requires special handling the memory unit to deal with the special address space. Comparing with the alternative design, we move the overhead from decoder to memory unit, and keep the contract between hardware and software unchanged. We prefer this design because it is more flexible. Although currently we need to change the microarchitecture in the processor core; but virtualization in the share cache or main memory would eliminate this limitation and allows for very flexible design.

There is also a tradeoff between hardware buffer overhead and communication overhead. The larger the buffer is, the less frequent the exception should be. The subsequent part of the paper will discuss the evaluation of this tradeoff.

3.3. MRC creation

Miss Rate Curve (MRC) is a quantification of cache size and cache miss rate, as the following figure shows. The x axis is the cache size measured in page number while the y axis is the miss rate. Given a target cache miss rate, we can select a minimized cache size configuration to satisfy the cache miss requirement. For example, if we want the cache miss rate to be about 2%, then we can set the cache at about 32 page sizes; further increasing the cache size is useless thought. For multiple program environments, this allows more efficient allocation policy without impacting the performance of each program. For power saving environment, extra cache space could be switched to low power mode to save power.

![Miss rate as a function of cache (memory) size](image)

In order to gain more flexibility in this process, we implement this work to the OS kernel. Our method is to maintain a LRU stack, which keeps track of the memory access stream by a FIFO order. Each new coming memory access address is pushed on the top of the stack. Before be pushed into the stack, we check whether it hits in the stack. If hit, we compute the distance of the entry to the stack top; if miss, we just assume a large enough value. We maintain a counter array $Hit[i]$ to record the situation of hit and miss. For each hit, supposed the distance is $i$, the corresponding counter is increased, $Hit[i] + 1$; for each miss, $Hit[\infty] + 1$. Here $\infty$ is set as the
size of the LRU stack. Thus, the miss rate along each cache size is computed as followed:

\[ MR(m) = 1 - \frac{\sum_{i=1}^{m} Hit[i]}{\sum_{i=1}^{m} Hit[i] + Hit[\infty]} \]

4. Experiment methodology

We use M5 [6] full system simulator and Linux 2.6.27 to perform this experiment. The M5 pipeline is augmented with a profiling stage in the cache module; currently we associate a small buffer with the cache to simplify implementation. More real implementation is to virtualize it in the lower memory hierarchy. A new exception type is defined and the corresponding system entry is added in the M5 internal processor file. The PAL instruction \( \text{wrent} \) and the table in PAL are augmented to support another exception. Linux is enhanced by another exception registered via PAL and the corresponding assembly entry is added. The stack layout is maintained exactly as ALPHA instruction \( \text{rti} \) requires as well.

We use SPEC2006 as benchmark to evaluate the LRU stack accuracy. The benchmarks are cross compiled to run on the M5-Linux without any change of the source code. The mechanism is transparent to the application, so source code is not required to be available. This facilitates the use in data center when the use applications are provide in binary code. This framework aims at runtime dynamic profiling and optimization without user intervenes.

5. Experiment result

5.1. LRU accuracy

We implement a 1024 entries LRU stack in the hardware to estimate the accuracy of the LRU stack. Results show that more than 90% of the hit/miss pattern could be accurately captured. The inaccuracy primarily comes from long distance reuse that exceed 1024 accesses, thus the stack could not keep the access while the real cache can. This approach is too restricted in two ways: pure hardware implementation and fully associative cache simulation. More advance cache simulation other LRU stack is maturely studied in [2]. It is very natural to extend the current work to simulate way associative cache. While our framework allows for software implement LRU stack which is more flexible to tradeoff accuracy with cache size. We will evaluate this tradeoff in later work.

5.2. Overhead estimation

The time overhead of this framework comes primarily from exception, which contains pipeline drain out, pipeline refill and exception handling. We roughly analyze the overhead by simulation cycles and compare with other exception types to demonstrate the low extra overhead of this framework.

Current M5 is an 8 issue processor and it assumes 13 cycles of exception responding. Thus the pipeline drains out and refill takes roughly 200 cycles. The exception handling is majorly dominated by the hardware buffer read, which depends on the size of the buffer. Given 2% of the cache miss, we anticipate a 200,000 memory accesses in each interval. Blocked aligned accesses mask out 5/6 of them for an 8 double word block size. So the handler needs to read roughly 40,000 times. The overall cycles required are at the level of 50,000 given an 8 issue processor.

The space overhead comes primarily from the hardware buffer, which is about 200kb. This is impractical on a chip, but virtualization would make such overhead reasonably acceptable in low level cache or main memory.

6. Summary

This paper introduces a cross layer mechanism combining the benefits of OS and CPU in shared
cache management, which dynamically captures the variant behavior of applications. According to our experiments, with simple LRU stack simulation, we are able to accurately capture more than 95% of cache hit/miss behavior; while the overhead, which is majorly the exception interface between CPU and OS, is considerably low. This work provides a framework of profile driven optimization which leverages the advantages of both OS and CPU in dynamic optimization, while it also explores feasibility and implementation of hardware software co-design which aims to maximize performance while minimize cost.

References


