Architectural Adaptation for Power and Performance

Weiyu Tang Alexander V. Veidenbaum Rajesh Gupta
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
E-mail: {wtang,alexv,rgupta}@ics.uci.edu

Abstract

Modern computer architectures represent design tradeoffs involving a large number of variables in a very large design space. Choices related to organization of major system blocks (CPU, cache, memory, I/O) do not work well across different applications. The performance and power variation across applications and against changing data set in a given application can easily be an order of magnitude.

Architectural adaptation provides an attractive means to ensure high performance and low power. Adaptable architectural components support multiple mechanisms that can be tailored to application needs. In this paper, we show the benefits of architectural adaptation for power and performance using the cache memory as an example. Dynamic L0 cache management selects either L0 cache or L1 cache for instruction fetch. It reduces average power consumption in instruction cache by 29.5% with only 0.7% performance degradation. Dynamic fetch size profiling changes cache fetch size at run-time to improve locality utilization. It improves average benchmark performance by 15%.

1. Introduction

Modern computer architectures represent design tradeoffs and optimizations involving a large number of variables in a very large design space. Even when successfully implemented for high performance, which is benchmarked against a set of representative applications, the performance optimization is only in an average sense. The performance variation across applications and against changing data set in a given application can easily be an order of magnitude. In other words, delivered performance can be less than one tenth of the system performance that the underlying hardware is capable of. A primary reason of this fragility in performance is that rigid architectural choices related to organization of major system blocks (CPU, cache, memory, I/O) do not work well across different applications.

Architectural adaptation provides an attractive means to ensure high performance and low power. Architectural adaptation refers to the capability of an architectural component to support multiple architectural mechanisms that can be tailored to application needs. Architectural adaptation has received a lot of attention recently with a drastic increase in VLSI complexity and transistor count as well as advances in reconfigurable logic. Making an architectural component adaptable typically involves the following steps:

- identify mechanisms that are important for performance/power and can be easily adapted in the architecture;
- determine the relationship between application behaviors and architectural mechanisms;
- design compiler-transformations or hardware predictions for architectural mechanism selection.

Architectural adaptation can be done at compile time or run-time. Run-time configuration can be controlled either by compiler-inserted instructions or by dynamic hardware prediction based on detected application behaviors. Dynamic hardware prediction is preferable, as it does not require code recompilation and Instruction Set Architecture modification.

The AMRM project at UC Irvine focuses on adaptation of the memory hierarchy and its role in latency and bandwidth management. In this paper, we describe how architectural adaptation can be used in improving system performance and power efficiency. This paper is organized as follows. Previous work on architectural adaptation is briefed in Section 2. Then we present two architectural adaptations as examples of how adaptive architectures can be useful. Dynamic L0 cache management for power efficiency is presented in Section 3, followed by dynamic cache fetch size profiling for high performance in Section 4. We conclude with Section 5.

2. Related work on architectural adaptation

There are a number of places where architectural adaptation can be used, for instance, in tailoring the interaction of processing with I/O, customization of CPU elements (e.g., splittable ALU resources) etc. Architectural adaptation is not a new idea. *Adaptive routing* pioneered by ARPANET in computer networks is applied to multiprocessor interconnection networks [3] to avoid congestion and route messages faster to their destination. *Adaptive traffic throttling* for interconnection networks [13] shows that "optimal" limit varies and suggests admitting messages into the network adaptively based on current network behavior .

Adaptive cache control or coherence protocol choices are investigated in the FLASH and JUMP-1 projects [6, 11]. *Adapting branch history length* [8] in branch predictors shows that optimal history length varies significantly among programs. *Adaptive adjustment of data prefetch length* in hardware is shown to be advantageous [4], while in [5] the prefetch lookahead distance is adjusted either in hardware or with compiler assistance.

Reconfigurable cache [12] enables the cache SRAM arrays to be dynamically divided into multiple partitions that can be used for different processor activities to improve performance. *Memory hierarchy reconfiguration* [1] dynamically detects application phase change and hit/miss tolerance. Then the boundaries between different levels of memory hierarchy are adjusted to improve memory hierarchy performance while taking energy consumption into consideration. *Adaptive cache line size* [14] exploits changing application locality with different cache line size to improve performance.

Pipeline gating [10] dynamically stalls instruction fetch to control rampant speculation in the pipeline. It reduces power consumption by reducing the number of wrong-path instructions.

3. Dynamic L0-Cache Management

As an example of adaptation for power management, let us consider design of the CPU-cache datapath for power. Due to its increasing size and high duty-cycle, cache is an important part of the power-performance strategies for modern high-performance processor designs. Further, to sustain the increasing instruction-level parallelism in modern processors, a very high utilization of the instruction memory hierarchy is needed. As a consequence, power consumption by the on-chip instruction cache is high. For instance, the power consumption by on-chip L1 instruction cache alone can comprise as high as 27% of the processor power[7]. To reduce the power consumption, a small "L0 cache," as shown in the left of Figure 1, can be placed between the L1 cache and the CPU to service the instruction stream to the CPU. A hit in the L0 cache can reduce power dissipation as a more power consuming L1 cache access is avoided. However, a miss in the L0 cache will incur additional L0 access latency for instruction fetch. Because the L0 cache size is small, the L0 cache miss rate is very high and the performance degradation can be more than 20% [9].

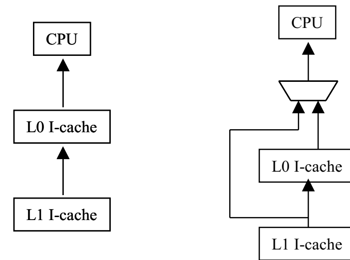

Figure 1. Memory hierarchy with L0 cache.

3.1. Adaptation Opportunities

An important reason for the performance degradation is the inflexibility of the memory hierarchy. For instance, here the L0 cache is always accessed first. Then on a L0 cache miss, the L1 cache is accessed. An adaptive datapath could enable, for some instructions that are unlikely to hit in the L0 cache, bypassing of the L0 cache to reduce the L0 miss latency without sacrificing power efficiency. A modified memory hierarchy with L0 cache bypass is shown in the right of Figure 1. This kind of bypass can be controlled either statically by compiler or dynamically by hardware prediction. In our research, we adopt the dynamic mechanism.

Generally, an instruction stream shows good spatial locality. It has been observed that if an instruction hits in the L0 cache, the remaining instructions in the same basic block is likely to hit in the L0 cache. Similarly, if an instruction misses in the L0 cache, the remaining instructions in the same basic block is likely to miss in the L0 cache. Thus we use the following prediction for dynamic L0/L1 cache selection:

- if current fetch hits in the L0 cache, then next fetch will go to the L0 cache;
- if current fetch misses in the L0 cache, then next fetch will go to the L1 cache.

Figure 2 shows the hardware support for dynamic L0/L1 prediction. L0 tag has dual roles, for determining whether an address hits in the L0 cache and for predicting the next fetch cache. Because of L0 tag's role in the prediction, we use decoupled data/tag arrays design for the L0 cache so that prediction can occur when current fetch is to the L1 cache.

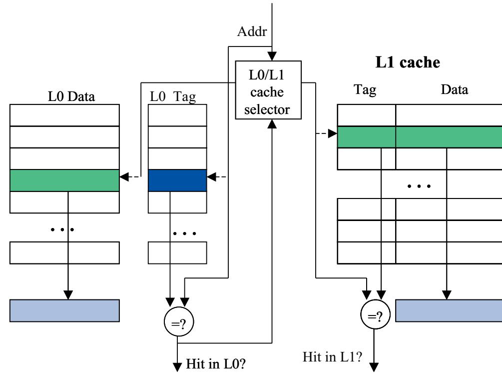

Figure 2. Dynamic L0/L1 prediction.

3.2. Experiments

We use the SimpleScalar tool set [2] to model an out-of-order superscalar processor. The processor and memory hierarchy parameters shown in Table 1 roughly correspond to those in current high-end microprocessors. The power parameters are obtained using Cacti [15] for the 0.18μm technology. For each SPEC95 benchmark, 100 million instructions are simulated.

Parameter	Value
branch pred.	combined, 4K-entry, 7-cycle mispredict. penalty
BTB	4K-entry, 4-way
RUU/LSQ	64/32
fetch/issue/commit width	4
int. ALU/MULT	4/2
flt. ALU/MULT	2/1
L0 I-cache	256B or 512B, 16B line direct-mapped, 1-cycle lat.
L1 I-cache	32KB, 32B line, 4-way, 1-cycle lat.
L1 D-cache	64KB, 32B line, 4-way, 1-cycle lat.
L2 cache	512KB, 64B line, 4-way, 8-cycle lat.
Memory	30-cycle lat.

Table 1. Processor and memory hierarchy configuration.

Figure 3 shows normalized execution time. The baseline system configuration for comparison has no L0 cache. The TRA bars are results for the traditional memory hierarchy with a L0 cache. The PRED bars are results for the memory hierarchy with L0 cache bypass based on dynamic L0/L1 prediction. For every benchmark, the execution time by PRED is lower than that by TRA. PRED is successful in reducing the performance penalty of L0 cache.

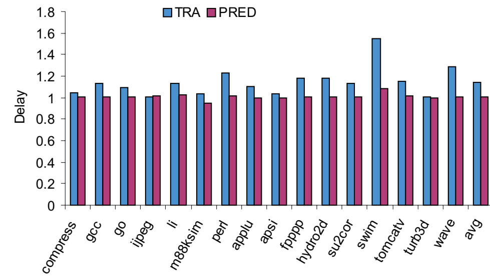

Figure 3. Normalized execution time.

For some benchmarks such as $m88ksim$, the normalized delay is lower than 1. Instruction fetches are delayed on miss-fetches in the L0 cache. Several instructions are committed during a miss-fetch cycle and branch history may be changed. This improves the branch prediction accuracy for some benchmarks. Comparing to 1 cycle L0 cache miss-fetch penalty, the branch misprediction penalty is 7 cycles. Thus more accurate branch prediction can result in performance improvement.

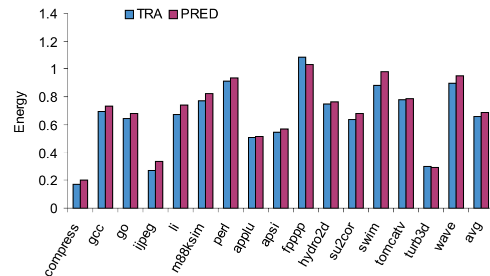

Figure 4. Normalized energy

Figure 4 shows normalized energy for the I-cache. The energy of PRED is close to the energy of TRA. PRED maintains the energy efficiency of the L0 cache. Average 29.5% energy savings are achieved using PRED with 0.7% performance degradation. Given such small performance degradation, PRED is suitable for high-performance processors.

Finally, Figure 5 shows normalized energy-delay product for the I-cache. The average energy-delay product of PRED is slightly better than that of TRA because the lower delay by PRED has compensated its higher energy. In addition, PRED is beneficial for energy-efficiency of the whole

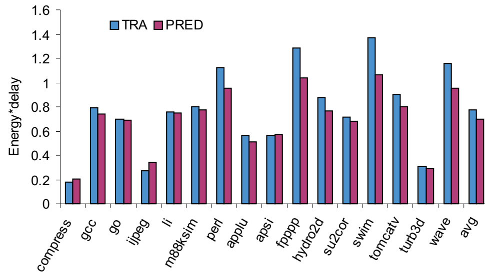

Figure 5. Energy-delay product.

system. With TRA, the energy-delay product of other processor components such as register files will increase dramatically because of the high delay of TRA. Hence energy efficiency of TRA for the I-cache may not translate into energy efficiency for the whole system.

4. Dynamic fetch size profiling

Caches are used to bridge the speed difference between the processor and the main memory. The usefulness of a cache lies in spatial and temporal locality of applications. A cache consists of multiple lines of equal size. Large line size benefits applications with good spatial locality and small line size benefits applications with good temporal locality. For a cache with a fixed line size, the determination of the line size is based on the spatial and temporal locality of average benchmarks. This fixed line size limits cache's ability in locality utilization.

As application locality changes over time, we can change the line size at run-time when locality change is detected. But actually changing line size at run-time is not trivial. The whole cache has to be flushed to ensure cache consistence.

In most cache designs, the "fetch size" is equal to the line size and one miss-fetch fills one cache line. However, the fetch size itself is an independent design parameter and can be a multiple of line size. On a miss-fetch, several cache lines can be filled simultaneously. Long fetch size can be used for applications with good spatial locality and short fetch size can be used for applications with good temporal locality.

4.1. Adaptation Opportunities

Although the optimal cache fetch size changes over time, a fetch size may stay optimal for an extended period. Thus we can use the optimal fetch size during a short period to predict the optimal fetch size during a long period.

We consider a fetch size optimal if it can result in minimal cache miss rate. It is impossible to measure miss rates for multiple fetch sizes at the same time. To predict the optimal fetch size, we can use the following assumption– *the locality in an application will not change dramatically during a short period*. Hence the optimal fetch size can be profiled in the following way:

1. select a list of fetch sizes;
2. apply each fetch size to a profiling interval and obtain the corresponding miss rate;
3. select the fetch size with the minimal miss rate as the fetch size for a stable interval.

There are performance penalties during fetch size profiling. Several fetch sizes are profiled and only one of them is the optimal. The miss rate during a profiling interval with a non-optimal fetch size may be much higher than the miss rate during a profiling interval with the optimal fetch size. Thus the stable interval should be much longer than the profiling interval to amortize the performance penalties during fetch size profiling.

The additional hardware support for dynamic fetch size profiling is: (a) a register to store current fetch size; (b) two registers to store interval lengths: one for the profiling interval, the other for the stable interval; (c) two counters, one for the profiling interval and the other for the stable interval; and (d) several registers to record miss rates for each fetch size.

4.2. Experiments

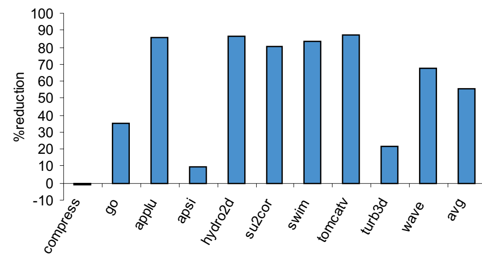

Figure 6. L2 cache miss rate reduction.

The experimental setup is the same as the setup for dynamic L0 cache management used in Section 3. Fetch size profiling can be done either for the L1 cache or for the L2 cache. We focus on the results for the L2 cache in this paper because L2 cache fetch size profiling can improve performance more than the L1 cache fetch size profiling can. The reason is that the L1 cache miss latency is much shorter than the L2 cache miss latency and most modern superscalar

processors can hide a large part of the L1 cache miss latency with other computations. Only SPEC95 benchmarks with non-trivial L2 miss rate are simulated. The profiling interval is 1,000 memory accesses and the stable interval is 100,000 memory accesses. The possible fetch sizes are: 64B, 128B, 256B and 512B.

Figure 6 shows L2 miss rate reduction. Out of 10 benchmarks, only $compress$ shows slight miss rate increase. The reason is that there are penalties in dynamic profiling and some predicted fetch sizes are not optimal. 6 benchmarks have more than 60% miss rate reduction. Dynamic fetch size profiling is effective in improving cache performance for most benchmarks.

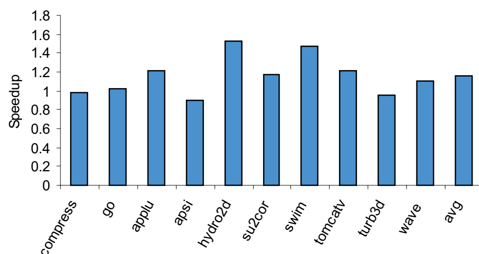

Figure 7. Speedup.

Figure 7 shows the speedup. We can see performance degradation for $compress$, $apsi$ and $turb3d$. The performance degradation for $compress$ is due to slight increase in the miss rate. The performance of $apsi$ and $turb3d$ degrades even though miss rate of them decreases. For these benchmarks, the traffic between the main memory and the L2 cache increases dramatically because of the use of large fetch sizes. Additional traffic increases the bus arbitration latency, which in turn increases the L2 cache miss-fetch latency. Longer miss-fetch latency degrades performance. For rest of the benchmarks, performance increases. The speedup ranges from 1.016 for go to 1.534 for $hydro2d$. The average speedup for all benchmarks is 1.15. Fetch size profiling can improve performance for most benchmarks.

5. Conclusion

Architectural adaptation can enable the application-specific customization for high performance and low power. We have demonstrated the benefits of architectural adaptation with two examples–dynamic L0 cache management and dynamic fetch size profiling. Dynamic L0 cache management can reduce average instruction cache power consumption by 29.5% and dynamic fetch size profiling can improve average performance by 15%. Going beyond these optimizations, our plans for future work include exploration of coordinated adaptation and its control through programming language extensions and compiler techniques.

Acknowledgments

This work is supported in part by the DARPA ITO under Grant DABT63-98-C-0045 and the DARPA PAC/C program.

References

[1] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Int'l Symp. Microarchitecture*, pages 245–257, 2000.

[2] D. Burger and T.Austin. The simplescalar toolset, version 2.0. Technical report, University of Wisconsin-Madison, 1997.

[3] A. Chien and J. Kim. Planar adaptive routing: low-cost adaptive networks for multiprocessors. In *Int'l Symp. Computer Architecture*, pages 268–277, 1992.

[4] F. Dahlgren, M. Dubois, and P. Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *Int'l Conf. Parallel Processing*, 1993.

[5] E. H. Gornish and A. Veidenbaum. An integrated hardware/software data prefetching scheme for shared-memory multiprocessors. In *Int'l Conf. Parallel Processing*, 1994.

[6] J. Kuskin et al. The stanford flash multiprocessor. In *Int'l Symp. Computer Architecture*, pages 302–313, 1994.

[7] J. Montanaro et al. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE Journal of Solid-State Circuits*, 32(11):1703–14, 1996.

[8] T. Juan, S. Sanjeevan, and J. Navaro. Dynamic history length fitting: a third level of adaptivity for branch prediction. In *Int'l Symp. Computer Architecture*, pages 155–166, 1998.

[9] J. Kin, M. Gupta, and W. Mangione-Smith. The filter cache: An energy efficient memory structure. In *Int'l Symp. Microarchitecture*, pages 184–193, 1997.

[10] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: speculation control for energy reduction. In *Int'l Symp. Computer Architecture*, pages 132–141, 1998.

[11] T. Matsumoto, K. Nishimura, T. Kudoh, K. Hiraki, H. Amano, and H. Tanaka. Distributed shared memory architecture for jump-1: a general-purpose mpp prototype. In *Int'l Symp. Parallel Architectures, Algorithms, and Networks*, pages 131–137, 1996.

[12] P. Ranganathan, S. Adve, and N. Jouppi. Reconfigurable caches and their application to media processing. In *Int'l Symp. Computer Architecture*, pages 214–224, 2000.

[13] S. Turner and A. Veidenbaum. Scalability of the cedar system. In *Int'l Conf. on Supercomputing*, pages 247–254, 1994.

[14] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Adapting cache line size to application behavior. In *Int'l Conf. on Supercomputing*, pages 145–154, 1999.

[15] S. Wilton and N. Jouppi. An enhanced access and cycle time model for on-chip caches. Technical report, Digital Western Research Laboratory, 1994.