Dynamically Adaptive Fetch Size Prediction for Data Caches

Weiyu Tang
Alex Veidenbaum
Alex Nicolau
Center for Embedded Systems
University of California Irvine
{wtang, alexv ,nicolau}@ics.uci.edu

Abstract

Cache line size has a significant impact on cache and overall CPU performance. This size is typically fixed at design time and may not be optimal for a given program or even within a program. Past attempts to achieve an effect of dynamic line size require complex hardware fetch size predictors. This paper proposes an adaptive fetch size predictor based on miss rate sampling. It requires little additional hardware and is straightforward to implement. Adaptive fetch size can be used at either L1 or L2 caches and achieves significant miss rate reductions in both cases. On average, the L2 adaptive fetch size cache results in highest overall performance improvements: 15% average speedup and up to 50% speedup in individual programs.

1. Introduction

Data cache performance in a modern high-performance processor is a function of a number of parameters, such as its size, associativity, line size, etc. These parameters are typically selected at design time based on implementation constraints, e.g. available silicon real estate and access time in a given technology, the benchmark suite behavior used to evaluate the implementation, and micro--architectural constraints. Once the final selection of these parameters is made their impact on cache performance is largely fixed. However, it may not be "optimal" for a given program or even within a given program.

There have been a number of attempts to design a cache in such a way that some of its parameters can be changed and to use prediction to dynamically "modify" such parameters to achieve higher performance [4, 11].

One parameter that has been attacked in such a manner is the cache block size. The reason is that an optimal block size is dependent on program behavior and its impact on performance is well known [4, 11]. This is typically a form of prefetching, attempting to bring additional data into the cache. However, the line size can also be effectively reduced in this way, either as a distinct fetching mechanism or a part of dynamic "line size" selection.

The argument is that an optimal line size for a given miss fetch is program, time, and reference dependent. Longer lines are desirable when spatial locality is high, shorter ones when it is low or when conflicts prevent the full use of a fetched line. Dynamic selection of the best line size for a given access is the problem addressed in this work.

In the context of this work, the dynamic line size will mean that a miss fetch will bring into the cache a number of cache lines between 1 and N. The number may be different for each miss fetch or for a group of miss fetches. Prediction of the right miss fetch size is a key contribution of this paper.

A smaller than usual line size is used for this cache, on the order of 8 or 16Bytes of data. This effectively allows a line size reduction, when compared to a 32- or 64B “standard” cache line. It is assumed that the fetch size predictor will set the current fetch size to 32- or 64Bytes, if it is the currently best size to use. Thus references that demand a standard line size will work equally well in the proposed system.

As will be seen in the related work section, similar types of fetch adaptivity have been proposed. The approach proposed in this work is more adaptive in terms of line size choices possible when compared to some of the existing proposals or significantly less complex when compared to more dynamic approaches.

The latter approaches use history-based prediction of fetch size, which requires a complex history-tracking mechanism for each reference. In contrast, the approach proposed here uses a simple sampling-based approach to find the best fetch size. In addition to hardware simplicity, this allows compiler participation. The compiler can select

the time to change the fetch size and/or decide on the new fetch size.

2. Related Work

Prior work in this area includes work on prefetching, macro- and super-blocks, adaptive line size, sectored cache design, and "spatial" footprint prediction.

Numerous prefetching algorithms have been proposed. They typically fetch a cache line or lines predicting future memory accesses. The prediction is based on access history and address prediction. For instance, a stream buffer [6] prefetches consecutive lines triggered by a miss. It stops when its FIFO buffer is filled. Buffer sizes of 4 or 8 cache lines are typical, resulting in multi-line prefetch. Prefetching schemes differ mainly in how they predict which block to fetch rather than in how many blocks to fetch. However, a variable-length prefetching mechanism was proposed by Dubois et al [3], which changes the number of blocks prefetched based on their utilization.

Sectored cache design [9] allows a subset of the words in a cache line to be fetched on a miss instead of a complete line. This works well if the line is larger than the optimal size for a given program and the rest of the line is not used. A good example of program behavior where this helps is strided array access with stride longer than a cache line. Sectored caches have been implemented in real systems, in particular by IBM.

Macro-blocks were proposed by Hwu et al [5] as a fixed-size super-line, typically 4x the size of a normal cache line. A detection mechanism is used to identify the use of multiple lines within a macro-block and fetch the entire macro-block on future misses. Based on the results of the detection, some lines are fetched with a normal size and others with a super size. The approach resulted in up to 15% improvement for SPECInt programs. Super-line detection requires a large hardware predictor to keep track of access history.

Baer et al [12] have investigated off-line algorithms for super-block detection and cache by-pass giving bounds on possible performance improvement.

Spatial footprint prediction mechanism was proposed by Wilkerson et al [7] for sectored caches. A subset of all sectors can be fetched on a miss, based on a prediction. The predictor used history-based information to record which of the sectors in a line were accessed and only those will be fetched on the next miss. This mechanism also requires a large hardware predictor to be effective.

An “adaptive line size” (ALS) cache was proposed by Veidenbaum et al [11]. It predicted the "line size" for the next fetch upon a line replacement. The line size had multiple possible values, from 8 to 512Bytes. The prediction was done for each line and stored at the next level in the memory hierarchy. The next predicted size could be 2x or 1/2x of the current size. The larger size prediction was based on the presence of an "adjacent" line in the cache. The smaller size prediction was based on partial line utilization as detected upon replacement. The ALS mechanism was shown to be effective for both L1 and L2 caches.

All of the prior schemes described above require a complex mechanism for detection and prediction of the next fetch size. The goal of this work is to define a simple mechanism requiring little hardware support to determine a future fetch size and to evaluate its performance impact. The details of the predictor design study are not reported here but can be found in [10].

3. Proposed Approach

The basic idea behind the adaptive fetch size (AFS) approach is similar to other dynamic mechanisms mentioned above: to fetch N “standard”-size cache lines into the cache on a miss. The difference is that a single fetch size is used on all misses, but this fetch size is periodically and dynamically adjusted. In addition, there is no restriction on what the next fetch size may be. For example, it is not limited to 2x or 1/2x of the current size.

A standard size cache line is defined here as a basic fetch unit that has its own tag in the cache. N is a dynamically variable number of fetch units that is set and periodically adjusted based on program behavior. The granularity of this adjustment is rather coarse. The AFS approach also defines a small basic fetch unit.

The AFS approach uses a very different fetch size predictor. It has low complexity and is based on sampling. The next fetch size prediction is determined by sampling different possible sizes and selecting the size with the lowest *observed* miss rate.

The predictor mechanism used in this study works as follows. Memory accesses are divided into “adaptivity intervals” of M memory references, where M can be between 100,000 and one million memory references. At the beginning of such an interval, fetch size sampling is used to determine the best fetch size N to use for the remainder of the current interval.

The sampling process uses a series of much smaller time intervals to search for the best fetch size. Each such interval, called a sampling interval, uses one fetch size and measures the resulting miss rate. The sampling interval size is K, K<<M, memory references. After all fetch sizes are tried the size leading to the best miss rate is selected.

Clearly, the size of the sampling interval, its relative size with respect to the adaptivity interval over which the predicted size is fixed, and the sampling algorithm itself have a major impact on performance. The evaluation of these parameters is beyond the scope of this paper, but can be found in [10].

For this approach to work, there has to exist a locally optimal fetch size, which is stable for a period of time. This has been observed to be the case by other researchers and in our experiments. The adaptivity interval has to be large enough to amortize any sampling inefficiencies. To achieve this, a sampling interval at least an order of magnitude smaller than the adaptivity interval is used.

The adaptive fetch size predictor is thus completely different than any of the past approaches mentioned above, including the ALS cache. It does not need to maintain any history information from interval to interval. Each interval is completely independent and builds its own history during the sampling phase. This property is the key to the simplicity of the predictor.

4. Performance Evaluation

The proposed approach was evaluated for a split L1 data cache and for a unified L2 cache. The baseline system used a Compaq Alpha 21264-like processor with a 4-wide issue and out-of-order execution. The baseline memory hierarchy used a 4-way set-associative, 32KB L1 data cache with a fixed-size 32B line. A 4-way set-associative, 512KB unified L2 cache used a 64B line. The memory hierarchy latencies were 3, 12, and 80 cycles, respectively, for the L1 data cache, the L2 cache, and the memory.

The evaluation was performed using the SimpleScalar 3.0 simulator [2] executing the 21264 binaries. SPEC95 benchmarks were used and simulated for 300M instructions after 100M instruction "warm-up". The reference data sets were used. While all SPEC95 benchmarks were evaluated, only results for benchmarks with non-negligible cache miss rates are presented.

The adaptive fetch size mechanism was used in the L1 data cache first, while using a standard L2 cache. In another set of experiments, a standard L1 data cache was used while the adaptive fetch size was utilized at the L2 cache (on data accesses only).

5. Results

The adaptivity interval size N used to evaluate the AFS caches was set at 100,000 cache misses for all experiments. The sampling interval M was one thousand cache misses. Notice that interval size is measured in cache misses, which are an order of magnitude lower than the number of memory accesses. Recall that benchmarks with negligible cache miss rates are not shown.

5.1. Cache Miss Rates

Let us start with the L1 adaptive fetch size cache. The possible fetch sizes were 16B, 32B, 64B, 128B, and 256B. The best fetch size is selected after five sampling intervals and used for the next adaptivity interval. All allowed fetch sized are sampled. An L2 cache with a 64B line was used.

Fig. 1 shows the impact of using the adaptive fetch size mechanism in the L1 data cache. The L1 miss rates for both the baseline configuration and the adaptive fetch size are shown. While some of the integer SPEC95 benchmarks have lower miss rates with adaptive fetch size, the baseline miss rate is very low to begin with. One low-miss rate benchmark, go, shows an increase in the miss rate when AFS is used.

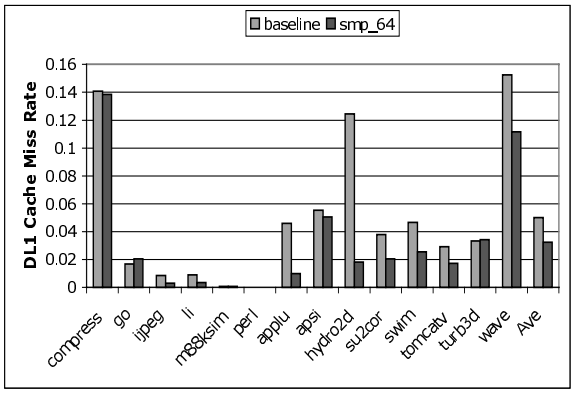

Figure 1. L1 Data Cache Miss Rates

The floating-point benchmarks show a much higher and more consistent miss rate reduction. The reduction range is between approximately 9 and 85%. It is clear that the adaptive fetch size mechanism works very well in this case.

The L2 cache with adaptive fetch size uses the same adaptivity and sampling intervals. It was used in conjunction with a 32B fixed line size in the L1 cache.

The possible fetch sizes, however, are 64B, 128B, 256B, and 512B. Fig. 2 shows the impact of adaptive fetch size on the L2 cache local miss rates. Again, the integer benchmarks are not really affected, except for go (which shows an almost 50% improvement). Two integer benchmarks show a negligible increase in miss rates. The floating-point benchmarks have very high local miss rates and show a significant miss rate reduction with adaptive fetch size, much higher than in the case of L1 cache. The majority of floating-point benchmarks achieve an 80+% miss rate reduction. It is clear that the adaptive fetch size impact at the L2 cache is more pronounced.

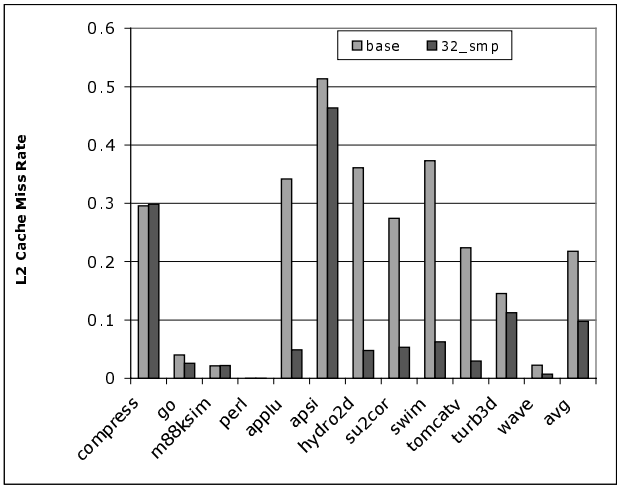

Figure 2. L2 Cache Miss Rates

It would appear that using the adaptive fetch size at both levels in the memory hierarchy would be advantageous as the results should be additive. However, combining the two is difficult and requires coordination. Otherwise it is possible, for instance, to have a larger L1 fetch size than the L2 fetch size. This would lead to additional misses, not to mention extra hardware complexity. The L2 cache appears to be the best choice for using AFS, if one had to choose only one level to use the adaptive fetch size at, given high L2 miss rates.

5.2. CPU Performance

A miss rate reduction alone does not necessarily guarantee an overall performance improvement as measured by the program execution time. This section presents the results of cycle-level timing simulation of a system with AFS caches.

The results shown in Fig. 3 are for two systems. One system uses fetch size adaptivity at the L1 cache and has a standard L2 cache with a 64B line. The second system uses fetch size adaptivity at the L2 cache, while the L1 cache uses a 32B line.

The first conclusion one draws from these results is that the fetch size adaptivity was not helpful at the L1 cache in most cases (hydro2d and applu are exceptions). The main reason is that the baseline L1 miss rates are not reduced enough to make a large difference. In addition, the out-of-order instruction issue masks some of the L1 cache miss latency.

Some programs experience a slowdown even with a miss rate reduction. This can be attributed to sampling overhead, which they experience while not gaining much from the L1 cache miss rate reduction.

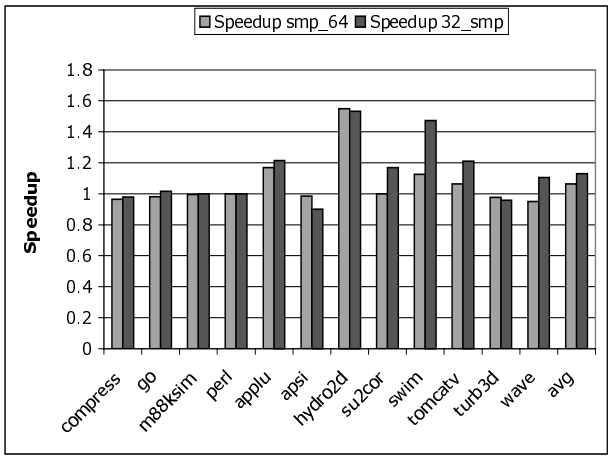

Figure 3. Execution Speedups

The results are significantly better for the L2 cache fetch size adaptivity. First, the baseline local miss rates shown in Fig. 2 are significantly higher than in the L1 cache. Second, the miss penalty for the L2 cache is much higher. Thus, even a program with a small baseline miss rate, such as wave, can experience a noticeable speedup.

On the other hand, the program with the highest L2 miss rate, apsi, does get any speedup. This happens for two reasons. First, the L2 miss rate alone does not tell the whole story and one has to consider the L1 miss rate as well. A program with one of the largest improvement, hydro2d, has one of the largest L1 miss rate of the suite. These misses are less costly when the L2 cache has a significantly reduced miss rate.

In general, a miss rate reduction does not directly translate into speedup. The memory stall component of the IPC has to be significant for the miss rate reduction to have an impact.

Overall, over half of all the benchmarks show a speedup when fetch size adaptivity is applied at the L2 cache. The average speedup for all benchmarks evaluated is approximately 15%. It can be as high as 50%. At the opposite end, two benchmarks exhibit a 5 to 10% slowdown when adaptive fetch is used.

6. Compiler-assisted fetch size adaptivity

The approach described above relied purely on hardware to determine the best fetch size to use. In addition, the sampling interval size and the adaptivity interval size were hardwired. A possible alternative is to involve the compiler in making all or some of these decisions.

The fixed adaptivity interval size implies that such an interval can start at any point in program execution. As programs spend most of their time executing within loops, one alternative is to have the compiler generate code to instruct the hardware when to start and when to end an adaptivity interval. For instance, upon entry into a loop nest would appear a good choice. Results for compiler-generated fetch size selection based on profiling have been reported in [8].

Another possibility is to compute a fetch size at compile time. The compiler can perform analysis of a loop nest and find a fetch size maximizing the cache performance. Compiler technology to do this for certain types of programs exists [1].

Using compiler-generated information requires a hardware API. This is beyond the scope of this paper, but some general statements can be made here. First, a hardware register holding current fetch size and an API for compiler to set it can be defined. Second, performance counters holding miss rates already exist. A restart signal for the sampling phase is all that is needed in this case.

7. Conclusions

This paper presented a new adaptive fetch approach for data caches. The main contribution of this work is the use of hardware-based miss rate sampling to determine the best fetch size to use in a time interval. The approach uses simple hardware and can be assisted by compilers.

Simulation results show that significant cache miss reductions can be achieved at both L1 and L2 caches for SPEC95 benchmarks with non-trivial baseline miss rates.

However, miss rate reduction does not always translate into speedup. Overall, the speedup is highest when fetch size adaptivity is applied at the L2 cache, both because of long miss latency and high L2 miss rates.

8. References

[1] P.D'Alberto, A. Veidenbaum, A. Nicolau, R. Gupta, "Static Analysis of Parameterized Loop Nests for Energy Efficient Use of Data Caches", *Compiler and Operating System for Low Power*, 2001.

[2] D. Burger, T.Austin, "The SimpleScalar toolset, version 2.0.", *TR-97-1342*, 1997, University of Wisconsin-Madison.

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

[4] K. Inoue, K. Kai, K. Murakami, "High Bandwidth, Variable Line-Size Cache Architecture for Merged DRAM/Logic LSIs", *Japanese IEICE Transactions on Electronics*, 1999, Vol. E81-C(9), pp. 1438-1447.

[5] T. L. Johnson, M. C. Merten, W. W. Hwu, "Run-time adaptive cache hierarchy management via reference analysis", *Int'l Symp.. Computer Architecture*, 1997, pp. 315-326.

[6] N.P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers", *Int'l Symp. Computer Architecture*, 1990, pp. 364-373.

[7] S. Kumar, C. Wilkerson, "Exploiting Spatial Locality in Data Cache Using Spatial Footprint", *Int'l Symp. Computer Architecture*, 1998, pp. 357-368.

[8] D. Nicolaescu, X. Ji, A. Veidenbaum, A. Nicolau, R. Gupta, "Compiler-Directed Cache Line Size Adaptivity", *Intelligent Memory System*, 2000, pp. 183-187.

[9] A.J. Smith, "Cache memories", *Computing Surveys*, 1982, pp. 473-530.

[10] W. Tang, A. Veidenbaum, A. Nicolau, R. Gupta, "Cache with Adaptive Fetch Size", *ICS-TR-00-16*, 2000, University of California, Irvine.

[11] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, X. Ji, "Adapting Cache Line Size to Application Behavior", *Int'l Conf. Supercomputing*, 1999, pp. 145-154.

[12] P. Vleet, E. Anderson, L. Brown, J.L. Baer, A. Karlin, "Run-time adaptive cache hierarchy management via reference analysis", *Int'l Conf. Computer Design*, 1999.