

Dynamic Parameter Tuning for Hardware Prefetching Using Shadow Tagging

Marius Grannæs and Lasse Natvig
Norwegian University of Science and Technology

Abstract—This paper presents a novel technique for dynamic selection of parameters for prefetching heuristics based on the use of shadow tag directories. Previous methods have been either static, made for a specific prefetching heuristic, or based on phase detection and tuning. The most flexible of these methods is phase detection and tuning. However, it has a serious drawback as it degrades performance while exploring the parameter space, as each configuration is tested on the running program. Our approach explores the parameter space using an extra structure called a shadow tag directory. This allows us to explore the parameter space without interfering with the running program, such that a larger parameter space can be explored without impacting performance. This paper examines the performance of this technique on tagged sequential prefetching, *czone/delta* correlation prefetching and reference prediction tables. In addition, we compare our results with a feedback directed approach. We show an overall 24% improvement over the best static sequential prefetcher and an 18% improvement versus feedback directed sequential prefetching on memory intensive SPEC benchmarks.

I. INTRODUCTION

The performance of general purpose microprocessors continue to increase at a rapid pace, but main memory has not been able to keep up. In essence, the processor is able to process several orders of magnitude more data than main memory is able to deliver on time. Numerous techniques have been developed to tolerate or compensate for this gap, including out-of-order execution, caches, prefetching and bypassing [9].

By utilizing prefetching, data that has not been referenced before can be inserted into the cache by analyzing the behaviour of the program and anticipating what data is needed in the future. Previous prefetching engines, such as sequential [17], reference prediction tables [4] (RPT) and *czone/delta* correlation [11] (C/DC), have static configurations. To provide the best average performance across a multitude of benchmarks a moderately aggressive prefetching configuration would be used. This leads

to performance degradation on some programs, as the prefetching is too aggressive, leading to memory bus congestion and cache pollution, while on other programs the full potential of prefetching can not be achieved because the aggressiveness is too low.

Consequently, there has been much interest in dynamic parametrization of prefetching heuristics. The idea is to adapt the prefetching heuristic to the running program by analyzing its behaviour. After analyzing the behaviour of the program, the parameters of the prefetcher (prefetching degree, prefetch distance, *czone* size etc) is adjusted accordingly.

A. Contributions

This paper investigates a general method for dynamically selecting prefetching parameters based on a shadow tag directory. Conceptually, a shadow tag directory is similar to a regular cache tag directory, which allows us to simulate the effects of altering the prefetcher configuration without interfering with the running program [7, 8, 14]. This method can be adapted to new prefetching heuristics with little effort. In addition, we show the benefit of dynamic parameterization with a sample heuristic that optimizes for performance by estimating prefetcher usefulness and bandwidth usage. The method is then evaluated on the memory intensive benchmarks in the SPEC2000 suite. We evaluate the technique combined with three different prefetchers, sequential, C/DC and RPT prefetching. We compare our scheme to no prefetching, their static counterpart, feedback directed prefetching and a perfect L2 (a cache that always hits). We show an overall 24% improvement over the best static sequential prefetcher and an 18% improvement versus feedback directed sequential prefetching on memory intensive SPEC benchmarks.

II. PREVIOUS WORK

A. Feedback Directed Prefetching

A few researchers have investigated dynamic parameterization of prefetching heuristics in the past.

A direct method is the feedback-based approach, which measures accuracy, timeliness and cache pollution caused by the prefetcher at runtime [19]. The values obtained are then fed into a state machine which in turn increases or decreases aggressiveness accordingly. Feedback directed prefetching estimates prefetcher accuracy by tagging each cache line with a single bit to signify that the line has been prefetched, but not yet accessed. Two counters are used; one that is incremented every time a prefetch is issued and another that is increased when a cache line that has the bit set is accessed. The ratio can then be used to estimate prefetcher accuracy. If the accuracy is high then the prefetcher aggressiveness is increased. Prefetcher timeliness and cache pollution is estimated in a similar way.

B. Tuning

A more general approach to dynamic reconfiguration is tuning. It has been successfully used in other areas as well, such as adapting the size of the issue queue to make it more power-efficient [3]. AC/DC prefetching is an extension of the static C/DC prefetcher that uses tuning to adapt to program phases [12].

Tuning works by detecting program phase changes by using instruction working sets, basic block vectors (BBV) and conditional branch counts [6]. When the program changes phase, it is likely that it will benefit from a reconfiguration of its resources [5]. The prefetching hardware will then search through the parameter space and select the best configuration for that phase. Each configuration is tried for a set amount of time (measured in clock cycles, number of L2 misses etc). When all the configurations have been explored, the best configuration is then used until the next phase change. However, as the parameter space grows, the time consumed by the search can become large. Thus, performance is degraded while the algorithm searches [15].

Tuning is usually implemented as a state machine with three states: stable, unstable and tuning [1, 5]. The stable state indicates that the best configuration is selected for that particular portion of the program. When a program undergoes a phase change it enters the unstable state. In this state the tuning algorithm simply waits while the program stabilizes. Once the program has stabilized, the tuning algorithm enters the tuning state, where it will search for the best selection of parameters.

Efficient tuning relies on good phase detection mechanisms to reliably detect phase changes and

identify similar phases. Phase change detection must be accurate, but allow for minor changes in program behaviour, so that it doesn't trigger tuning unnecessarily.

C. Shadow Tag Directories

A shadow tag directory is functionally similar to a regular cache directory. However, the shadow tag directory does not have a corresponding data array. Its purpose is to simulate, at runtime, the result of having a different prefetching configuration. Both directories receive the same memory access stream, and is manipulated accordingly.

Dybdahl et al. used this technique to augment a cache replacement policy for chip multiprocessors [8]. They used shadow tags for dynamically switching between their cache replacement policy and the traditional LRU policy, based on their relative performance. A similar work by Dybdahl et al. used shadow tags to improve cache partitioning in chip multiprocessors [7]. Simultaneously, Qureshi et al. used shadow tags¹ to switch between a cache replacement policy that optimizes the amount of memory level parallelism and the traditional LRU policy [14].

Both papers illustrate that the shadow tag directory only needs to contain about 20 sets to be effective. Qureshi establishes this by use of an interesting statistical proof. Although the two papers differ in the selection of the 20 sets, the results are similar.

III. METHODOLOGY

A. Shadow Tag Controlled Prefetching

We propose a new approach to dynamic tuning of prefetching heuristics by using a shadow tag directory. Our architecture has two levels of cache and main memory as shown in Figure 1. The main prefetching engine is connected to the second level cache. All second level cache accesses are also propagated to the shadow tag directory. The shadow tag directory is similar in configuration to the L2 cache, but does not hold any actual data. It is connected to another prefetch engine running with an alternative configuration.

The controller is a small unit responsible for reconfiguring the prefetch engines based on data gathered from the two tag directories. After 2000² cache misses have occurred in the real L2 cache, the performance of the two configurations are compared. If the results

¹Qureshi et al. uses the term auxiliary tag directory.

²Earlier research has by Dybdahl[7] has shown that comparing every 1000 misses is preferable. We found that an interval of 2000 L2 misses provided more robustness.

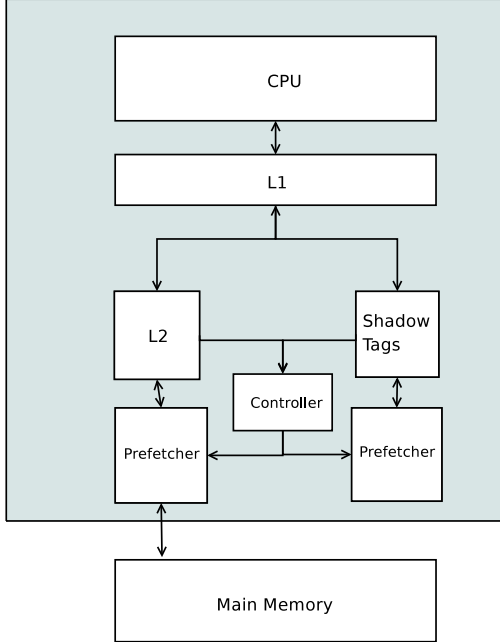


Figure 1. The proposed architecture. The L1 cache is connected to both the L2 cache and the shadow tag directory. The controller evaluates the performance of the two configurations and reconfigures the prefetchers accordingly.

from the shadow tags are better than the results from the real L2 cache, then the configuration of the shadow-prefetcher is applied to the main prefetcher.

In either case, the shadow-prefetcher is then reconfigured, so that another configuration is explored.

The shadow tag directory is similar to the L2 tag directory, every access by the processor is inserted into the tag directory. In addition, the miss address stream fed into the shadow-prefetcher which uses this information to generate (simulated) shadow-prefetches. These shadow prefetches are then inserted into the shadow tag directory, so that the shadow tag directory reflects the state it would have been in if the prefetches would have been issued.

To estimate the hardware overhead of using a shadow tag directory we have used the cache modelling tool Cacti [16]. In 65nm technology the tag directory of the L2 cache used in this paper uses 0.12 mm^2 , while the L2 itself uses 7.48 mm^2 . A shadow tag array would thus increase the size of the L2 by 1.6%.

However, both Dybdahl and Qureshi [7, 8, 14] have shown that it is only necessary to replicate 20 of the 512 sets for accurate predictions. In other words, the area requirements for the shadow tags

can potentially be reduced to around 0.005 mm^2 , or about an increase in L2 size of 0.06%.

Additionally, it is possible to use the shadow tag directory for other performance enhancing techniques such as increasing the amount of memory level parallelism [14] and temporal locality of the cache [7, 8]. Thus this hardware cost can be amortized across several techniques.

In this paper we have used a full-sized shadow tag directory for our simulations, replicating all the sets in the original L2 cache.

B. Prefetch Configuration Selection Heuristic

The overall goal is to increase performance in terms of IPC (Instructions Per Clock Cycle). It is not possible to directly measure the effect on IPC by using shadow tags as they contain no data. By using the methods described by Srinath [19] it is possible to estimate prefetcher accuracy, timeliness and cache pollution in addition to the number of cache misses and cache hits.

When a prefetch is issued, a counter is increased (indicating the number of prefetches issued) and a prefetched-bit is set in the corresponding cache line. This bit is already present when using sequential prefetching, and thus causes no additional overhead. The first time a cache line with this bit set is referenced by the program, the bit is cleared and another counter (indicating the number of successful prefetches) is increased. By dividing the two numbers, we get an estimate of the prefetchers accuracy.

A simple method for estimating memory traffic is used. We record the number of L2 misses that would have occurred if the shadow tag directory configuration was used. By dividing the number of clock cycles elapsed by the amount of L2 misses, we get an estimate of the amount of memory traffic that would occur if the shadow configuration is used.

Initially, the real prefetcher is set to a prefetching degree of zero (or off). The shadow prefetcher is set to a random prefetching degree (0-16) and a random prefetch distance (0-16) (and a random czone size (4KB-4MB) if applicable). After 2000 L2 misses has occurred, the two configurations are evaluated.

We use a fitness function to evaluate both configurations:

$$F = Hits - \frac{Late}{4} - \lfloor 2^{BW-T} \rfloor \quad (1)$$

Our heuristic has three components. First, we use the number of cache hits. Since we evaluate the two configurations after 2000 L2 misses, this gives us

indirectly a measurement of the hit-ratio. The second component is the number of late prefetches. Late prefetches are prefetches that have been issued, but have been issued too late to fully cover the complete memory latency. This component is estimated by using a prefetch bit in the MSHRs in a similar manner as the method used by Srinath [19]. To decrease the number of late prefetches a prefetcher needs to issue prefetches with a larger *prefetch distance*. The last component is a simple function depending on the bandwidth usage. We estimate bandwidth usage by using the number of cache misses that have occurred and the time elapsed, this number is denoted *BW* (Bandwidth Usage). To ensure that this component does not become dominant when there is ample bandwidth available, we subtract a fixed threshold value from this number (*T*).

If the fitness of the shadow tag directory exceeds that of the real cache, the configuration of the shadow tag prefetcher is adopted to the real prefetcher. Then a new random configuration for the shadow tags is chosen and the process is repeated.

C. Experimental Setup

For the evaluation of this proposed architecture we have extended the SimpleScalar [2] simulator with shadow tag directories, prefetching and a new model for main memory that simulates contention. Our main memory model models DDR2 memory and accounts for split transactions, open/closed pages, burst mode, multiple channels and pipelining. The setup can be found in table I. We used the reduced data set [10] SPEC2000 benchmarks [18]. This datasets allows us to run each benchmark to completion. To compensate for the small datasets, the L2 cache is relatively small compared to the aggressive core. This was done to force more misses in the L2 cache and further stress the memory subsystem.

Out of the 26 benchmarks in the suite, we have selected the 10 most memory intensive applications measured in terms of the number of memory accesses per instruction. On the remaining 16 benchmarks in the SPEC2000 benchmark suite we observe no significant performance improvement or degradation. This is due to the entire data set of the benchmark fitting inside the L2 cache, thus giving little opportunity for prefetching. Thus these 16 benchmarks are omitted in the remainder of this paper.

IV. RESULTS

We have examined our technique on three different prefetching heuristics, tagged sequential, *czone/delta*

correlation (C/DC) and reference prediction tables (RPT). We compare our dynamic scheme to no prefetching, the best static parameter selection, feedback directed prefetching and a perfect L2 (a L2 that never misses). The original implementation of feedback directed prefetching included a method for controlling the insertion policy of prefetched cache-lines. This part of feedback directed prefetching has been omitted to make it easier to compare the two methods.

1) *Sequential Prefetching*: In figure 2 we show the performance of the 10 most memory-constrained benchmarks in the SPEC2000 benchmark using sequential prefetching. First, we observe that the static configuration leads to degraded performance on both Mcf and Bzip2 compared to the case of no prefetching, while our shadow tag scheme only has a minor regression on Bzip2. Furthermore, we see significant improvements on the Ammp benchmark. Shadow tag prefetching gets an IPC of 0.55 while feedback directed prefetching gets an IPC of 0.46. This is due to the very low accuracy (4%) of sequential prefetching, which leads feedback directed prefetching to not increase aggressiveness to the required level. In total we observe a 18% increase in harmonic mean compared to feedback directed prefetching, 24% improvement over the static configuration and 48% over no prefetching. In addition, we observe that the largest regression compared to feedback directed prefetching is 5% (Bzip2).

2) *C/DC prefetching*: In figure 3 we show the performance of the techniques on *czone/delta* correlation prefetching. Feedback directed prefetching has no method for controlling *czone* size directly, so we have used the best static value for the *czone* size. We observe that the static configured C/DC prefetcher shows regressions on Applu and Facerec. However, shadow tags performs better than both feedback directed and static prefetching. In addition, the performance on Ammp is increased even further. Overall, we observe an increase in harmonic mean by 58% compared to no prefetching, 5.6% improvement versus static prefetching and a 5% improvement versus feedback directed prefetching. It is worth noting that we observe a significantly higher prefetching accuracy on C/DC prefetching than sequential prefetching.

3) *RPT prefetching*: Reference Prediction Tables is a very robust technique. Our results for this prefetching technique is shown in figure 4. It has a very high prefetch accuracy (more than 90%), but lower coverage than the other prefetchers. Even with a static configuration we observe no regressions ver-

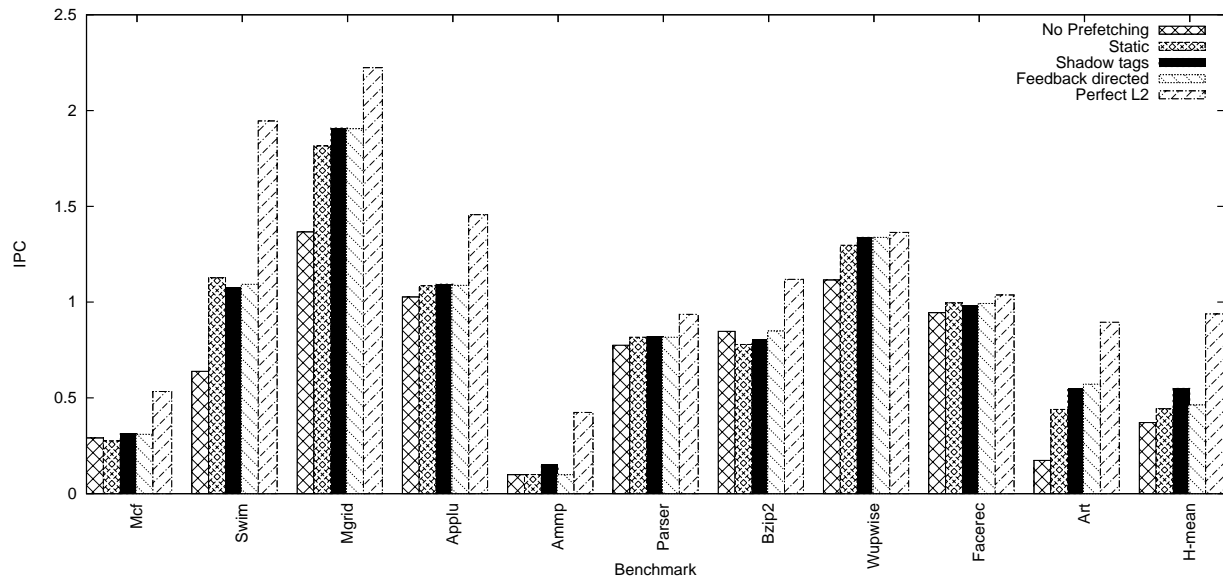


Figure 2. Performance of dynamic parameter selection on static prefetching.

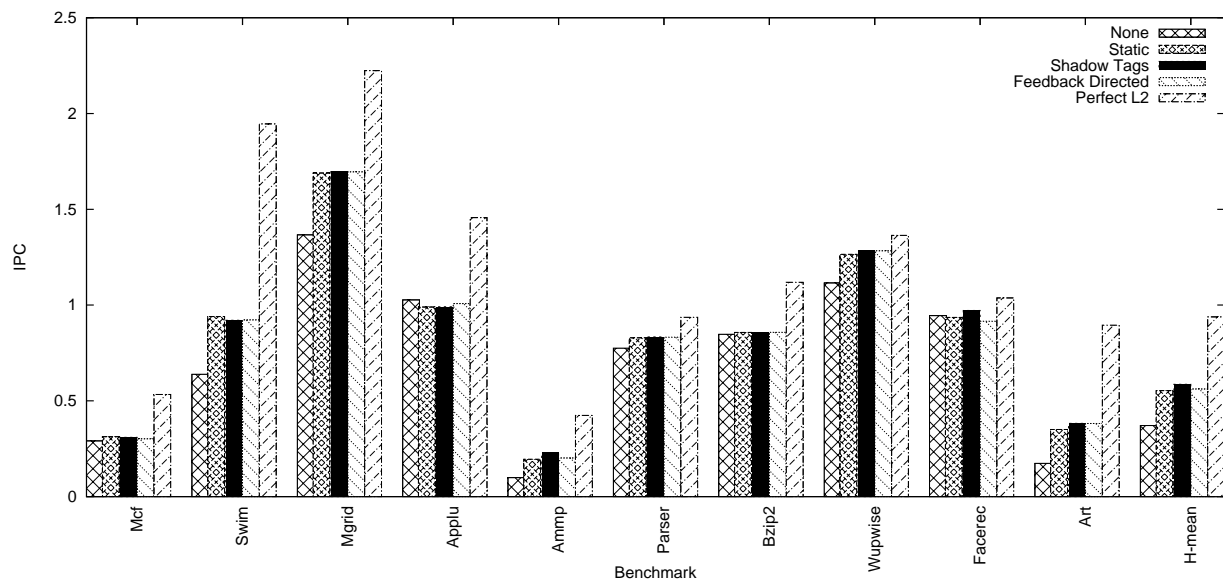


Figure 3. Performance of dynamic parameter selection on C/DC prefetching.

Clock Frequency	4 GHz
Processor Width	4 instructions/cycle
Register Update Unit	64 instructions
Load/Store Queue	32 instructions
Fetch Queue	16 instructions
Functional Units	4 ALUs, 1 Integer Multiply/Divide 4 FPUs, 1 Floating Point Multiply/Divide
Branch Predictor	Combined, Bimodal 4K entry table, 2-level 1K table, 10 bit history table, 4K Chooser, 4-way 512 entry BTB, 15 cycles miss predict penalty
TLB (D & I)	128 entry full associative, 30 cycle miss penalty
Level 1 D-Cache	8K 4-way, 64B blocks, LRU 2 cycle latency
Level 1 I-Cache	8K 4-way, 64B blocks, LRU 2 cycle latency
Level 2 Cache	512K 8-way, 128B blocks, LRU, 7 cycle latency
Main Memory	160 cycle latency, max bandwidth 9GB/s

Table I
THE SIMULATION PARAMETERS USED WITH SIMPLESCALAR.

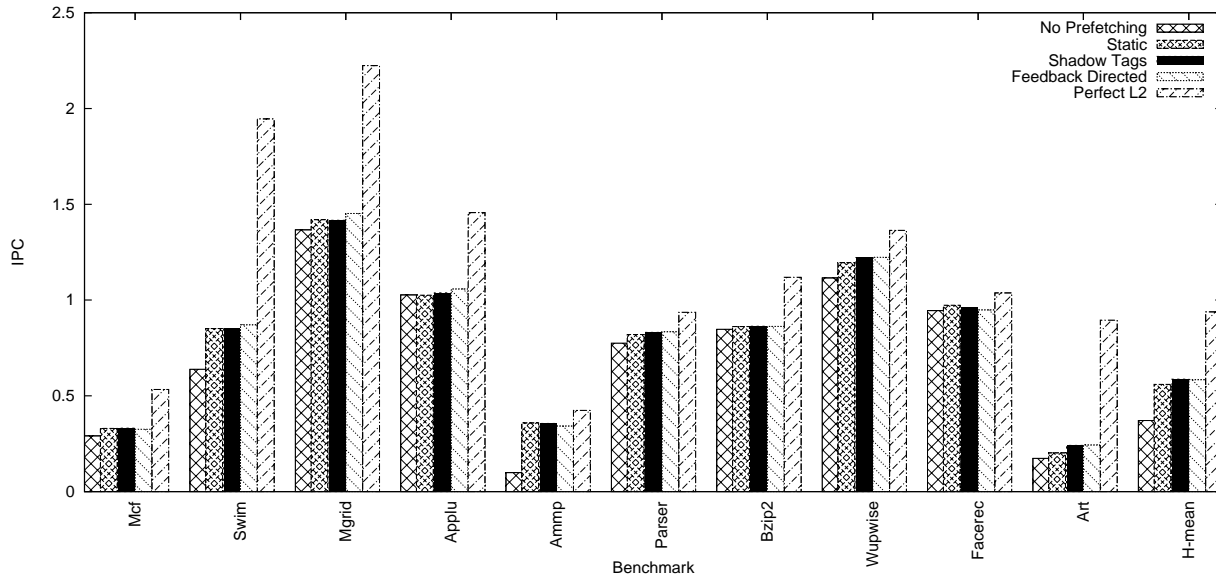


Figure 4. Performance of dynamic parameter selection on RPT prefetching.

sus the case of no prefetching. The dynamic schemes perform as well or better than the static configuration in all but two cases: Ammp and Facerec. We observe an increase of 58.2% in harmonic mean by using shadow tags versus no prefetching. However, the gains versus static prefetching is smaller, only 4.8%. The difference versus feedback directed is minimal, 0.48%. It should be noted that RPT prefetching requires that the address of the load-instruction is included with every memory request, thus making it

expensive to implement in hardware.

A. Bandwidth Usage

Prefetching will necessarily increase bandwidth requirements (unless the prefetcher is 100% accurate). Figure 5 shows the bandwidth usage for each benchmark. The numbers have been normalized to the case of no prefetching. Overall, sequential prefetching requires a substantially more bandwidth than C/DC or RPT prefetching. Furthermore, the bandwidth

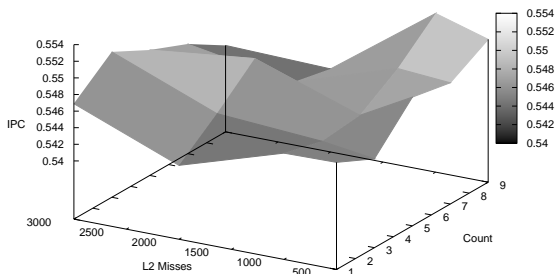


Figure 6. Performance of shadow tag prefetching as a function of parameters to the heuristic.

requirements of Amp and Bzip2 stands out as excessive. This can be justified, especially on Amp where performance is increased by a considerable amount. On average shadow tag prefetching requires 11% more bandwidth for sequential prefetching compared to a static configuration, while it requires 0.7% more on C/DC prefetching and 3.4% more on RPT prefetching. However, it should be added that our heuristic is designed to use whatever bandwidth is available.

B. Sensitivity Analysis

In this section, we look at some of the parameters that we have used in our heuristic. In figure 6 we look at how often the reconfiguration occurs (*Misses between reconfiguration*) and the number of consecutive checks where the shadow tags must outperform the real cache for the configuration of the shadow tags to be adopted (*Count*). These two parameters do not have a significant impact on the harmonic mean of IPC. The difference in IPC is only 2.5%. It should be noted that configurations such as 500 L2 misses and a count of 1 is more unstable. Some benchmarks will benefit greatly, while others gets reduced performance, however, the speedups and slowdowns evens up over multiple benchmarks.

Figure 7 shows the harmonic mean of IPC for the benchmarks as a function of the bandwidth threshold. The number on the X-axis represents the average number of clock cycles between memory accesses. A low number denotes that a configuration with a high bandwidth usage will be penalized less. This graph

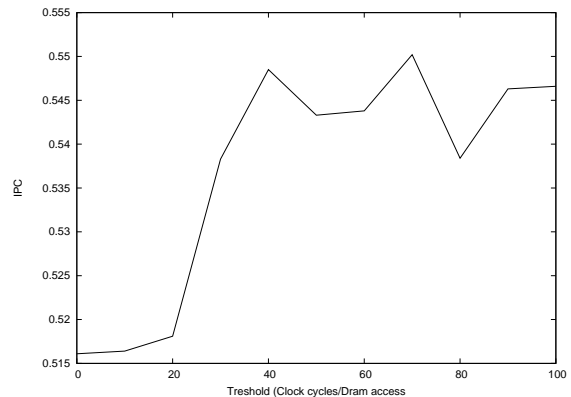


Figure 7. Performance of shadow tag prefetching as a function of the bandwidth threshold parameter.

shows that if the heuristic is not penalized for using too much bandwidth then performance drops. It is worth noting the sharp increase in IPC in the interval 20 to 40. This interval represents a bus utilization around 80%.

V. DISCUSSION

A. Parameter Space Exploration

In this paper the configuration to be tested on the shadow tags were chosen randomly across the whole parameter space. We have also experimented with using hill climbing to explore the parameter space. Hill climbing only looks at small changes in the configuration and selects the best configuration out of those tested. A known problem with hill climbing is that it can get stuck in local optima and thus not find the global optimum. We believe that such local optima are not likely to be stable as the program runs. However, because hill climbing “moves” slower across the parameter space, we observed a 18% decrease in performance on some benchmarks that were too short for the hill climbing technique to converge on a good solution.

There are other approaches that could be used, such as genetic algorithms and simulated annealing [13].

B. Clearing the Shadow Tags

Initially, we were concerned that not clearing the shadow tag directory between two configurations would pollute subsequent measurements. We examined the effect of copying the contents of the real

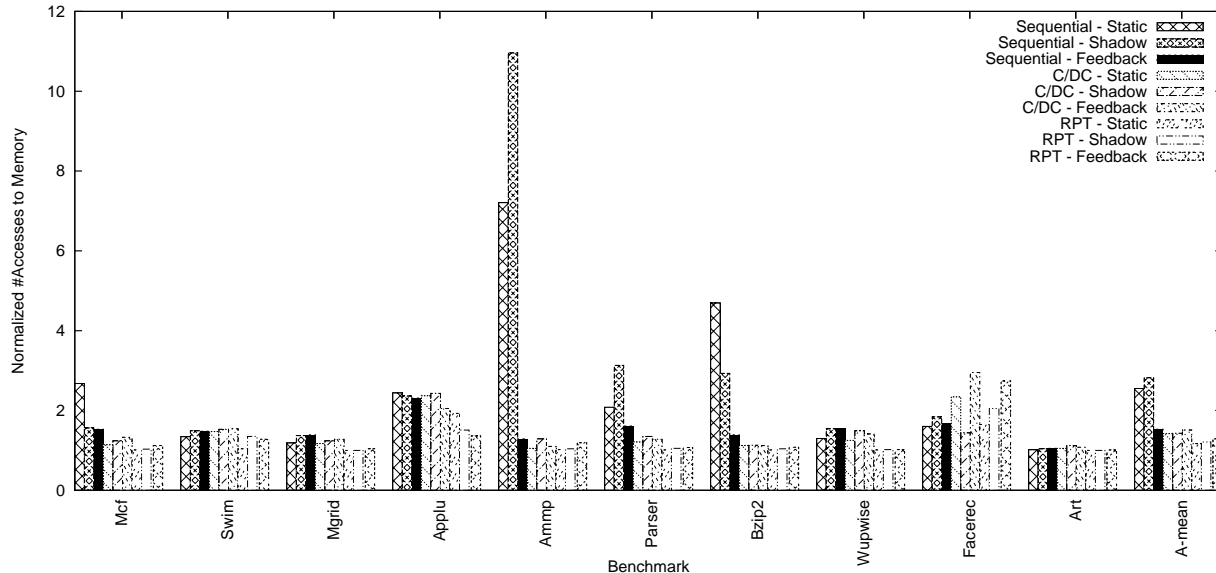


Figure 5. Number of main memory accesses for different combinations of prefetching heuristics and parameter selection methods. Values are normalized to no prefetching.

tag directory to the shadow tag directory after each reconfiguration. We found that copying had little impact on performance (we observed only a 0.1% improvement by using a copying function). In addition, the cost of such a mechanism would be prohibitive both in terms of area and power.

VI. CONCLUSION

In this paper we present a novel technique for dynamic parameterization of prefetching heuristics. By using this technique we observe an overall improvement in IPC by 24% over the static configuration and a 18% improvement over feedback-directed prefetching. However, the relatively large improvements seen on single benchmarks are equally important. In addition, we observed no regressions against the case of no prefetching, which makes the method robust and only one regression against the case of a static prefetcher.

In terms of cost, the L2 is increased in size by 1.6% for an overall gain of 24%. It might be possible to reduced this to about 0.06% if the methods by Dybdahl et al. and Qureshi et al. can be used with prefetching. The shadow tag directory can be used for other purposes as well, such as optimizing memory level parallelism and enhancing the cache replacement policy, thus amortizing this cost over several performance enhancing techniques.

REFERENCES

- [1] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 245–257, 2000. ISBN 1-58113-196-8. doi: <http://doi.acm.org/10.1145/360128.360153>.
- [2] D. Burger and T. M. Austin. SimpleScalar toolset 3.0b, 2003. <http://www.simpleScalar.com>.
- [3] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi. An adaptive issue queue for reduced power at high performance. In *Power-Aware Computer Systems: First International Workshop, PACS 2000*, 2000.
- [4] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *Computers, IEEE Transactions on*, 44:609–623, May 1995.
- [5] A. Dhodapkar and J. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings. 29th Annual International Symposium on Computer Architecture*, pages 233–244, 2002.
- [6] A. S. Dhodapkar and J. E. Smith. Compar-

- ing program phase detection techniques. In *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, 2002.
- [7] H. Dybdahl, P. Stenstrom, and L. Natvig. A cache-partitioning aware replacement policy for chip multiprocessors. In *Proceedings of IEEE International Conference on High Performance Computing*, 2006.
- [8] H. Dybdahl, P. Stenstrom, and L. Natvig. An LRU-based replacement algorithm augmented with frequency of access in shared chip-multiprocessor caches. *Computer Architecture News*, 35, 2007.
- [9] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 3rd Edition*. Morgan Kaufmann Publishers, 2003. ISBN 1-55860-724-2.
- [10] A. KleinOsowski and D. J. Lilja. MinneSPEC: A new spec benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.
- [11] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. *Micro, IEEE*, 25:90–97, Jan. 2005.
- [12] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: An adaptive data cache prefetcher. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, pages 135–145, 2004.
- [13] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.
- [14] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 167–178, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2608-X. doi: <http://dx.doi.org/10.1109/ISCA.2006.5>.
- [15] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings. 30th Annual International Symposium on Computer Architecture*, pages 336–347, 2003.
- [16] P. Shivakumar and N. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical Report 2, Compaq Western Research Laboratory, August 2001.
- [17] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982. ISSN 0360-0300. doi: <http://doi.acm.org/10.1145/356887.356892>.
- [18] SPEC. Spec 2000 benchmark suites, 2000. <http://www.spec.org>.
- [19] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. Technical report, University of Texas at Austin, May 2006. TR-HPS-2006-006.