Combining Local and Global History for High Performance Data Prefetching

Martin Dimitrov   Huiyang Zhou

School of Electrical Engineering and Computer Science
University of Central Florida
{dimitrov,zhou}@eecs.ucf.edu

Abstract

In this paper, we present our design for a high performance prefetcher, which exploits various localities in both local cache-miss streams (misses generated from the same instruction) and the global cache-miss address stream (the misses from different instructions). Besides the stride and context localities that have been exploited in previous work, we identify new data localities and incorporate novel prefetching algorithms into our design.

In this work, we also study the (largely overlooked) importance of eliminating redundant prefetches. We use logic to remove local (by the same instruction) redundant prefetches and we use a Bloom filter or MSHRs to remove global (by all instructions) redundant prefetches. We evaluate three different design points for the proposed architecture, trading off performance for complexity and latency efficiency. Our experimental results based on a set of SPEC 2006 benchmarks show that we improve performance (over 1.6X for our highest performance design point) for various processor configurations, which model different cache capacity and memory access bandwidth.

1. Introduction

Data prefetching has been recognized as a promising way to overcome the adverse impact of the ever increasing gap between memory access and processor speeds. Previous proposed prefetchers have exploited data locality, mainly stride-based [2][4] and context-based [3] locality in address streams to predict future reference addresses and then to prefetch them into caches before the data are required by the processor. However, as reported in a recent study [11], the performance improvements of the latest data prefetchers are limited, not much beyond the classical stride-based stream buffers [4]. In this paper, we propose a high performance data prefetcher, which explores both local cache-missing address stream, meaning cache-missing addresses generated by the same instruction, and global cache-missing address stream, meaning addresses generated by all loads and stores, for regular address patterns. Besides stride and context locality, this prefetcher incorporates new localities that have not been exploited in previous works.

Furthermore, we highlight the importance of eliminating redundant prefetches. We propose to use logic to eliminate local (from the same instruction) redundant

prefetches and a Bloom filter or MSHRs to remove global (by all instructions) redundant prefetches. Our experimental results based on a set of SPEC 2006 benchmarks show that our design achieves significant performance improvement (over 1.6X for our highest performance design point).

In summary, the main contributions of this paper include:

- A high performance data prefetcher that exploits various data localities in both local and global cache missing address streams;
- New data localities including global strides, most common local strides, and local scalar patterns;
- Evaluating three different design points including a simple, complexity and latency efficient design.
- Advocating for L1-cache data prefetchers despite the traditional wisdom that favors L2-cache prefetchers;
- Recognizing the importance of redundant prefetches, which has been largely overlooked in previous works.
- An adaptive scheme to turn off the prefetcher when it is no longer beneficial.

The remainder of the paper is organized as follows. Due to limited space, in Section 2 we discuss only the most closely related works and their limitations. In Section 3, we present our newly identified data localities. Our data prefetcher architecture as well as the storage cost is presented in Section 4. The experimental results are shown in Section 5. Finally, Section 6 concludes the paper.

2. Related Work

Due to its importance, data prefetching has been an active research topic in processor design. Next-line prefetching and its improvement, tagged prefetching [12] are classical ways to leverage spatial locality in data streams. Stride-based prefetching schemes [2][4] detect the stride pattern (a, $a+d$, $a+2d$, …) in the address stream and issue prefetches based on the dynamically captured strides. Context prefetching or correlation prefetching [3] detects the correlation between cache miss addresses (e.g., a, b, a, b, …) and issues prefetches based on the previously recorded correlated addresses. Because of the large address range, it usually requires a large buffer to capture context correlation in address streams. One effective improvement over address correlation is to capture correlation in delta (i.e., difference between

consecutive addresses). This way, both stride and correlation locality can be detected effectively. Delta correlation was proposed for TLB prefetching [5] and adapted for cache prefetching [8][9].

Global history buffer (GHB) [8] provides an efficient way to maintain the most recent cache misses and can be used to implement flexible prefetching algorithms. The structure of a generic GHB prefetcher is shown in Figure 1. The GHB is organized as a circular FIFO buffer with each entry maintaining an address and a pointer. The addresses in GHB are managed as many linked lists and the index table provides a pointer to the head of each linked list. Dependent upon the key to the index table, various histories-of-interest can be reconstructed from the GHB. For example, if the key is the program counter (PC), the local miss address streams can be reconstructed from the global history. A recent study [11] shows that the GHB prefetcher achieves the best performance among the ten prefetching mechanisms under the study.
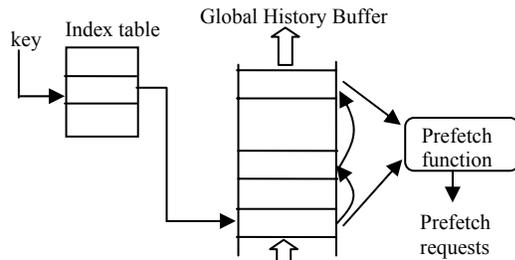
Figure 1. Global history buffer prefetching [8][9]

Although GHB prefetching has many desirable features, it has two major weaknesses. (a) Since the GHB maintains the global miss history and is shared by all misses, it can be resource inefficient when dealing with a burst of misses from few static instructions. For example, a few frequently missed loads/stores with perfect stride patterns may pollute all the GHB entries although few entries are needed to detect the strides. (b) It requires sequential operations to traverse a linked list to reconstruct an address history of interest from the GHB. Although such latency penalty may not be an issue for an L2-cache prefetcher due to the high L2-cache miss latency, such sequential operations make it less attractive for an L1-cache prefetcher. In this work, we combine GHB with local delta buffers (LDBs) to provide faster access and better resource efficiency (see Section 4).

3. Novel Data Localities in Address Streams

Following the convention used in value prediction research [6][13], we use *local* history to refer to the addresses generated by dynamic instances of the *same* instruction and *global* history to refer to the addresses generated by all instructions. As discussed in Section 2, existing works on data prefetching have exploited stride and/or context locality in local and/or global address streams.

In this paper, the following new data localities have been identified.

- **Global stride**

This locality exists when there is a constant stride between addresses of two different instructions. For example, in the following global address stream: X, $X+d$… Y, $Y+d$… where (X, Y …) is the local address history from an instruction I while ($X+d$, $Y+d$ …) is the local address history of an instruction J. In this case, even if there is no exploitable pattern in the local address histories of I or J, the address stream of the instruction I can be used to prefetch data for instruction J once the global stride is detected. The global stride between two dynamic instructions' destination values was observed in [13]. Address-value delta (AVD) [7] can also be viewed as a form of global stride in address streams since the load value is used as the address for its dependent load instruction.

GHB provides an efficient way to detect the global stride locality: using the PC as a key, each linked list comprises one local address history. The global strides can be computed as the difference between each entry in the linked list and the entry next to it, as shown in Figure 2.
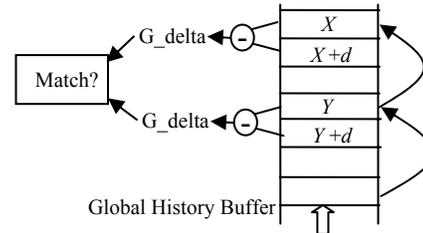
Figure 2. Computing global strides using GHB

- **Most common stride**

This locality appears when a constant stride pattern is disrupted from time to time with some irregular addresses. For example, in the following delta address (the difference between two consecutive addresses) stream: d, x, d, y, d, z… although there exists a common stride d, it can not be detected with existing stride or context-based approaches. In our experiments, we observed such locality in several benchmarks and we devise a simple way to detect it in local address streams (see Section 4). In [10] the authors propose to prefetch the minimum delta from the delta buffer, rather than the most common as we propose.

- **Scalar**

Again, considering the delta address stream, the scalar locality exists in the following case: d, $2d$, $4d$, $8d$, $16d$, etc. Such locality is a direct result of the code that we observed in the benchmark *mcf*. In the function *replace_weaker_arc*, the indices of some array accesses are generated using the code "cmp *= 2" and then the
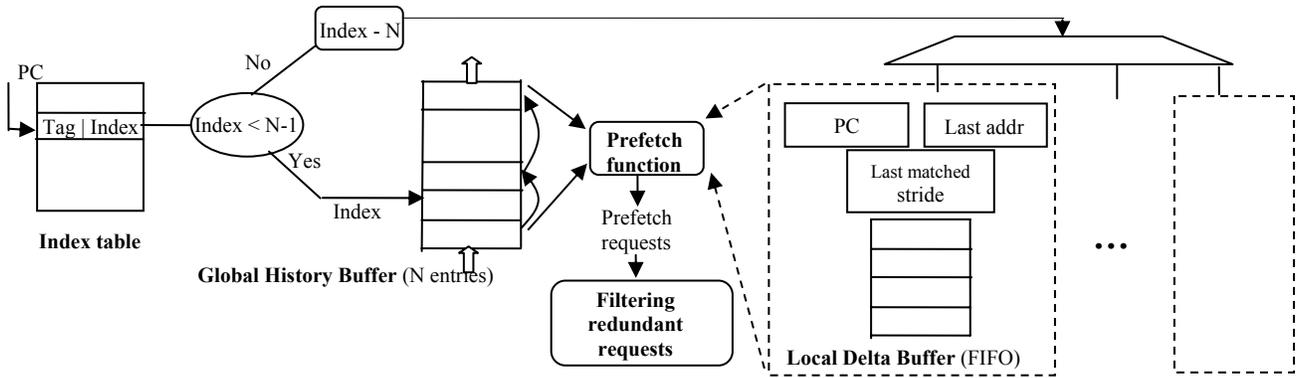
Figure 3. The structure of the proposed data prefetcher

array is accessed using "if(new[cmp-1].flow < new[cmp].flow)".

Although it is not difficult to detect such locality, we found that the scalar pattern seems to be a rare case except in *mcf*. Even in *mcf*, the pattern varies due to control flow and the delta stream becomes *d*, 2*d*, 4*d+m*, 8*d+m*, etc. Therefore, we exclude the scalar locality in our current data prefetcher.

4. Proposed Data Prefetcher

4.1. Architecture

Our proposed prefetcher exploits data localities in both local and global address streams. Unlike the GHB prefetcher, we use a structure named local delta buffer (LDB) for instructions with strong locality in their local histories and a GHB for global address stream for other instructions. The overall structure of our proposed prefetcher is shown in Figure 3. Next, we discuss each of the key components in detail.

- **Index table**

Similar to a GHB prefetcher, the index table in our design is a cache-like structure indexed by the PC. Each entry in the index table has three fields: the tag, least recently used (LRU) counter, and index. If the index field is greater than the number of entries (*N*) in the GHB, the value (index – *N*) is used to access one of the LDBs. Otherwise, the index field points to an entry in the GHB, which is the beginning of the linked list of the addresses generated by the same instruction with the corresponding tag.

- **Global history buffer (GHB)**

In our design, the GHB operates in the same way as in [8][9] except that not all the miss addresses are sent to the GHB. If an index field in the index table points to one LDB, the miss address will be sent to the LDB accordingly. Since the addresses in the GHB are linked using the PC, each linked list is a local address stream.

- **Prefetch function**

The prefetch function in our design implements the following prefetching algorithms.

(1) Delta correlation

To capture delta correlation, a delta buffer is included in the prefetch function, which keeps the delta information when a linked list is traversed in the GHB. Then, a match of two consecutive deltas is searched in the delta buffer using the same approach as in [9]. If there is a match, *8* prefetches will be issued according to the delta pattern. As consecutive delta matches indicate strong locality, the delta buffer is copied to one of the LDBs (the least recently used one). The index field is then updated correspondingly so that subsequent addresses from the same instruction will be sent directly to the LDB.

(2) Simple delta correlation

If there is no match of two consecutive deltas found in the delta buffer, *4* prefetches will be issued if there is a match of the last delta in the delta buffer. For example, if the delta buffer contains (*a*, *x*, *y*, *z*, *a*, *m*, *n* …) with *a* being the latest delta, the prefetch requests will be (*last address + z*), (*last address + z + y*), (*last address + z + y + x*), and (*last address + z + y + x + a*).

(3) Global stride

When the linked list is traversed in the GHB, the global strides are computed as described in Section 3. If there is a match found, prefetches can be issued accordingly.

(4) Most common stride and next-line prefetch

This function applies to the delta buffer originating from the LDBs. If there is no delta correlation found in the local history, next-line prefetch and most common stride (see LDB discussion next) are used to generate two prefetches.

- **Local Delta Buffer (LDB)**

An LDB is a FIFO structure and contains a local delta address stream. It also has a PC field as the tag and an LRU counter for replacement. The last address is maintained to calculate the latest delta. The prefetch functions are the same as those used for GHB except that there is no need to traverse the GHB to compute the delta

buffer. The "*last matched stride*" field is updated when there is a match found during delta and simple delta correlation computation. It implies that the stored stride at least appears twice in the delta buffer. This field is designed to approximate the most common stride discussed in Section 3 and is used only when there is no delta correlation.

In our proposed design, multiple LDBs are used and the idea is to allocate a LDB for each of the most frequently missed load or store instructions. This way, each miss address of those instructions will go to LDB directly without polluting the GHB. In addition, the latency of accessing the GHB sequentially (i.e., linked list traversal) is eliminated for prompt prefetch request generation in the common case.

4.2. Filtering of Redundant Prefetches

Our prefetcher prefetches data into L1 data cache (D-cache). The reasons are two-folds. The first is that L1 cache miss address stream provides much stronger locality than L2 miss addresses, which translates into higher prefetch accuracy. The second is that prefetching data into L1 D-cache can eliminate the L2 cache access latency. We quantify the impact of prefetching into the L1 cache vs. prefetching into the L2 cache in Section 5.

When used as L1-cache prefetcher, each L1 miss (also L1 hits if the block is prefetched) will access the prefetcher and potentially invoke prefetch requests. The requests from different misses may overlap with each other and result in wasted bandwidth. Redundant prefetches may be triggered by the same instruction, which repeatedly issues the full prefetch degree upon each prefetcher access (prefetch a,b,c,d then prefetch b,c,d,e etc.). Redundant prefetches may also be triggered by a different instruction, which accesses the same set of cache lines. Thus even if individual instructions do not issue any redundant prefetches, the prefetches may be redundant with respect to those issued by other instructions. Therefore, a filtering mechanism is necessary especially when the bandwidth support is limited.

Two different mechanisms are used in our proposed design. First, we use simple logic to eliminate some *local* redundant prefetches. Second, we employ a set of prefetch MSHRs or a bloom filter [1] to eliminate the remaining redundant ones. The simple logic here targets at both constant stride and repeating context patterns. One "*confidence*" bit is added to each LDB. If a constant stride or a strong delta correlation is detected (meaning 3 consecutive matches in delta correlation), this bit is set. If this bit is set, subsequent accesses to the LDB will only issue one unique prefetch request. For example, an LDB detects a constant stride and issues prefetches ($a+d$, $a+2d$, $a+3d$ … $a+8d$) where a is the current miss address. After the confidence bit is set, subsequent accesses (e.g, address $b=a+d$) will result in a single prefetch $b+8d$ ($=a+9d$) rather than ($b+d$, $b+2d$ …). Each prefetch address which

passes the logic filter, probes the MSHR or the Bloom filter. If there is a Bloom filter hit, it is likely that we have already issued this prefetch, and we discard it. On a Bloom filter miss, we issue the prefetch. The Bloom filter may generate false-positive matches, and those will result in an incorrectly dropped prefetch. To limit false-positive matches, we reset the Bloom filter periodically (every $n/4$ filter accesses, where n is the number of filter entries). The MSHRs will not result in false positive matches, however they are much more expensive the implement.

Besides the structures described above, we also use a set of counters to monitor the effect of our prefetcher. In a training phase, we turn off prefetching if the prefetching block address satisfies the condition: (*block addr* % 4 = = 2). Then, we periodically (every 1 million cycles) compare the miss rate from these lines that are not affected by the prefetcher with the miss rate from other lines to see whether the prefetcher is beneficial. If not, we can turn off the prefetcher.

4.3. Design Space Exploration

In this work, we propose and evaluate three different design points of the architecture described in Section 4.1 and Section 4.2. The first design point (GHB-LDB-1) is our highest performance design, which aggressively exploits all the localities as described above. This design also uses MSHRs to remove redundant prefetches without any false-positive matches (see Section 4.3).

The second design point is similar to the first, however it is scaled down in terms of storage requirements (a smaller GHB table is used) and complexity (Bloom filter is used instead of MSHRs). We call this design point GHB-LDB-2, and the purpose is to demonstrate that even with a significantly smaller storage budget, we can still achieve very high performance.

Our third design point is meant to be the most *complexity* and *latency* efficient. It uses a single-level prefetch table indexed by load/store PC. Each entry in the prefetch table is a fixed size, local delta buffer (LDB) as described in Section 4.1. Thus each table entry maintains the last several deltas (strides) for a given load/store instruction as well as the last miss address for computing the new delta. Since the LDB-only design does not maintain global history information, it cannot prefetch global strides. However, the LDB prefetcher can detect the delta correlation, simple delta correlation and most common stride patterns as discussed in Section 4.1. In this design, upon a prefetch table hit, the prefetch function will search for delta and simple delta correlation. If no match is found, then a next line prefetch is generated. This design is simple and latency efficient, because it does not require a link-list traversal or other complex logic. It also eliminates the problem with a GHB based design, where a burst of misses from only a couple of load/store instructions may pollute the entire GHB. Despite of its simple design, the LDB-only prefetcher is

able to achieve impressive speedups when prefetching for the L1-cache, as shown in our experiments in Section 5.

4.4. Storage Cost

For the competition, we submitted three design points: GHB-LDB-1, GHB-LDB-2 and LDB-only as described above. The configuration and the storage cost of all three versions are summarized in Table 1. In both GHB-LDB versions, we use a 256-entry 8-way set-associative index table. Assuming 32-bit processors, it requires 256 * (27 bit tag + 3 bit LRU + 8 bit index to GHB) = 9728 bits. For the GHB, each entry takes a 32-bit address and a $\log_2(N)$ bit pointer, where N is the number of entries in the GHB.

Each LDB has a 7-entry delta buffer (the eighth delta is calculated using the current address and the '*last address*' field) and each delta is 32 bits. Therefore, each LDB takes (7*32 + 32 bit PC + 32 bit last address + 32 bit 'last matched stride' + $\log_2(M)$ bit LRU) where M is the number of LDBs in the design. The prefetch function has a 32-entry delta buffer (32*32) and three temporal registers (32*3) for delta matching, a total of 1120 bits.

In the LDB-only version we use a 64-entry 8-way table. Each table entry is an LDB, thus the storage is the same as above. The only difference is that we use 24 bit deltas.

The prefetch MSHRs maintain outstanding prefetches at the cache block level. Since the block size if 64 bytes, each MSHR costs (26 – index + LRU bits).

The adaptive control of the prefetcher requires several counters. We collect miss rates (3 counters for L1, L2, and the region in L2 not affected by prefetching) every 1 million cycles. We also use prefetch bit to get the number of successful prefetches in L1 cache. A total of 100 bits are allocated for those counters. The LDB-only prefetcher does not use the adaptive control in our experiments.

Table 1. A summary of storage cost of two submitted versions

Storage Cost	GHB-LDB-1	GHB-LDB-2	LDB-only
Index Table	256-entry 8-way 9728 bits	256-entry 8-way 9728 bits	64-entry 8-way
GHB	192 entry 192 * (32+8) = 7680 bits	128 entry 128 * (32+7) = 4992 bits	N/A
Prefetch Func.	1120 bits	1120 bits	1120 bits
Prefetch MSHR	256-entry 8-way 256*(21+3)=6144 bits	N/A	N/A
Bloom filter	N/A	2048 + 8-bit reset counter	4096 + 9-bit reset counter
LDBs	16 LDBs 16*(7*32+32+32 +32+5)=5200 bits	16 LDBs 16*(7*32+32+32 +32+4+1)=5200 bits	64 LDBs 64*(7*24+32+32 +3+1)=15104 bits
Counters	100 bits	100 bits	N/A
Total	29972 bits (3.7kB)	23196 bits (2.9kB)	20329 bits (2kB)

5. Experimental Results

We model the proposed prefetcher using the simulation framework for the 1st Championship data prefetching contest (DPC-1) [14]. We used *gcc* 4.1.2 on a 32-bit X86 machine to compile a set of memory intensive SPEC 2006 benchmarks. For each benchmark, the trace was generated by skipping the first 40 billion instructions and recording the next 100 million instructions. The performance improvements (compared to no prefetching) are measured for three processor configurations according the rules of DPC-1 and the results are shown in Tables 2, 3 and 4 for the three submitted design points.

Table 2. The speedups from prefetcher (GHB-LDB-1)

Speedup	bzip2	lbm	mcf	milc	omnetpp	soplex	xalan	Gmean
Config1	1.07	2.89	2.65	1.97	1.13	1.54	0.99	1.61
Config2	1.08	2.98	1.90	2.83	1.10	1.46	0.97	1.60
Config3	1.02	2.98	1.88	2.83	1.11	1.48	1.37	1.67

Table 3. The speedups from prefetcher (GHB-LDB-2)

Speedup	bzip2	lbm	mcf	milc	omnetpp	Soplex	xalan	Gmean
Config1	1.07	2.88	2.53	1.95	1.17	1.48	0.97	1.59
Config2	1.08	2.77	1.84	2.78	1.12	1.47	0.94	1.57
Config3	1.01	2.80	1.83	2.78	1.13	1.51	1.37	1.65

Table 4. The speedups from prefetcher (LDB)

Speedup	bzip2	lbm	mcf	milc	omnetpp	Soplex	xalan	Gmean
Config1	1.07	2.83	2.38	1.91	1.13	1.47	0.89	1.54
Config2	1.08	2.92	1.85	2.48	1.09	1.47	0.85	1.53
Config3	1.04	2.91	1.84	2.48	1.10	1.55	1.30	1.63

Besides remarkable performance improvements, some interesting observations have can be made from our experiments. First, the simple LDB approach is able to capture most of the performance benefit of the more complex GHB-LDB scheme. Thus capturing local delta correlations and filtering redundant prefetches account as the major contributors to performance improvement. On the other hand, we found that for the GHB-LDB design around 90% of all the prefetches are issued from LDBs instead of the GHB. Since LDBs eliminate the need to traverse the linked list in GHB, the latency of prefetch generation is effectively reduced. Second, a relatively small number of LDBs (16) is enough for either version to achieve the most performance enhancement. In our submissions we scale the GHB-LDB-1 to use more storage budget.

As discussed in Section 4.3 in contrast to conventional wisdom, we advocate prefetching into the L1 data cache, instead of at lower levels of the cache hierarchy. Table 5 shows the performance results of our best performing GHB-LDB-1 configuration when applied to the L2 data cache. We can see that the speedup is significantly limited due to the less accurate prefetches.

Table 6 shows the impact on performance when redundant prefetches are not eliminated. For configuration 1 (unlimited bandwidth) we actually achieve speedup. However, for configurations 2 and 3 (limited bandwidth – maximum 1 request every per cycle from L1 to L2 and maximum 1 request every 10 cycles from L2 to memory)

the performance is severely impacted, even observing slowdowns.

In Table 7 for comparison, we show the performance of the original GHB proposal. We implement GHB to the best of our knowledge. Detect only delta context strides (match of last two deltas) as described in the original paper. We also do not filter redundant prefetches. Again, we can see that while GHB performs reasonably well for configuration 1, when the bandwidth is limited it suffers a lot due to redundant prefetches.

Table 5. Prefetching for the L2-Cache (GHB-LDB-1)

Speedup	bzip2	lbm	mcf	milc	omnetpp	soplex	xalan	Gmean
Config1	1.05	2.22	1.90	1.89	1.15	1.39	0.96	1.44
Config2	1.06	0.96	1.11	1.68	1.02	1.27	0.94	1.13
Config3	1.00	0.96	1.11	1.67	1.02	1.20	1.29	1.16

Note*: The large drop in performance when prefetching into the L2 cache, can be also be attributed to a problem using the prefetch MSHRs. When we use a BloomFilter instead of MSHRs, the geomentric mean of the speedups is 1.40, 1.51, 1.57 for configurations 1, 2 and 3 respectively. The benchmark lbm in particular has speedups of: 1.93, 2.40, 2.40 for the three configurations when using the bloom filter.

Table 6. No filtering of redundant prefetches (GHB-LDB-1)

Speedup	bzip2	lbm	mcf	milc	omnetpp	Soplex	xalan	Gmean
Config1	1.07	2.89	2.64	1.97	1.18	1.55	0.99	1.62
Config2	1.07	0.51	0.91	2.72	1.11	1.15	0.96	1.07
Config3	0.96	0.51	0.91	2.70	1.11	1.18	1.42	1.12

Table 7. Original GHB approach, prefetching into the L1-Cache and no filtering of redundant prefetches

Speedup	bzip2	lbm	mcf	milc	omnetpp	Soplex	xalan	Gmean
Config1	1.06	2.89	2.40	1.96	1.10	1.30	0.83	1.50
Config2	1.06	0.66	0.94	2.15	1.07	1.11	0.77	1.04
Config3	1.03	0.65	0.94	2.15	1.08	1.15	1.11	1.09

Lastly, we note that the compiler, which is used to compile the SPEC benchmarks and generate the traces, plays an important role. When changing to another *gcc* version (4.2.3), the performance varies significantly (both with and without prefetching) compared to the results using *gcc* 4.1.2. Nevertheless, our proposed prefetcher still achieves significant speedups (1.5~1.7X) in this case.

6. Conclusions

In this paper, we present our design for an L1 data cache prefetcher. It exploits various data localities in both local and global address histories and achieves remarkable performance improvements. In this work, we also emphasize the importance of removing redundant prefetches to reduce bandwidth demand.

7. Acknowledgements

8. References

[1] B. Bloom, "Space/Time trade-offs in hash coding with allowable errors," *Communications of the ACM*, 1970.

[2] J. Fu and J. Patel, "Stride directed prefetching in scalar processors", *MICRO-25*, 1992.

[3] D. Joseph and R. Grunwald, "Prefetching using Markov predictors", *ISCA* 1997.

[4] N. Jouppi, "Improving direct-mapped cacge performance by the addition of a small fully-associative cache and prefetch buffers", *ISCA* 1990.

[5] G. Kandiraju and A. Sivasubramaniam, "Going the distance for TLB prefetching: an application-driven study", *ISCA*, 2002.

[6] M. Lipasti, C. Wilkerson, and J. Shen, "Value locality and load value prediction", *ASPLOS* 1996.

[7] O. Mutlu, K. Kim, and Y. Patt, "Address-value delta prediction: increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns", *MICRO-38*, 2005.

[8] K. Nesbit and J. E. Smith, "Prefetching with a global history buffer", *HPCA* 2004.

[9] K. Nesbit, A. Dhodapkar, and J. E. Smith, "AC/DC: An adaptive data cache prefetcher", *PACT* 2004.

[10] S. Palacharla and R.E Kessler, "Evaluating Stream Buffers as a secondary cache replacement", ISCA 1994.

[11] D. Perez, et. al., "Microlib: A case for the quantitative comparison of micro-architecture mechanisms", *MICRO-37*, 2007.

[12] A. Smith, "Cache memories", *Computing Surveys*, 1982.

[13] H. Zhou, J. Flanagan, and T. Conte, "Detecting global stride locality in value streams", *ISCA* 2003.

[14] The 1st JILP Championship data prefetching contest (http://www.jilp.org/dpc)