Data Access History Cache and Associated Data Prefetching Mechanisms

Yong Chen [1] Surendra Byna [1] Xian-He Sun [1, 2]

chenyon1@iit.edu sbyna@iit.edu sun@iit.edu

[1] Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616, USA

[2] Computing Division, Fermi National Accelerator Laboratory, Batavia, IL 60510, USA

ABSTRACT

Data prefetching is an effective way to bridge the increasing performance gap between processor and memory. As computing power is increasing much faster than memory performance, we suggest that it is time to have a dedicated cache to store data access histories and to serve prefetching to mask data access latency effectively. We thus propose a new cache structure, named Data Access History Cache (DAHC), and study its associated prefetching mechanisms. The DAHC behaves as a cache for recent reference information instead of as a traditional cache for instructions or data. Theoretically, it is capable of supporting many well known history-based prefetching algorithms, especially adaptive and aggressive approaches. We have carried out simulation experiments to validate DAHC design and DAHC-based data prefetching methodologies and to demonstrate performance gains. The DAHC provides a practical approach to reaping data prefetching benefits and its associated prefetching mechanisms are proven more effective than traditional approaches.

Categories and Subject Descriptors

C.4 [Performance of Systems]: Design Studies

General Terms

Performance, Design, Verification

Keywords

Data access performance, Memory performance, Data prefetching, Prefetching simulation, Cache memory

1. INTRODUCTION

While microprocessor performance improved by 52% a year until 2004 and has been increasing by 25% from then, memory speed is only increasing by roughly 9% each year [9]. The performance disparity between processor and memory keeps expanding. Deeper memory hierarchies were introduced to bridge this gap [9]. Each memory level closer to the processor is smaller and faster than the next lower level. The rationale behind memory hierarchy design is the principle of data locality, which states that programs tend to reuse data and instructions which are accessed recently (temporal locality) or to access those items whose addresses are close to one another (spatial locality). However, when applications lack locality due to a working set size larger than the cache and/or non-contiguous memory accesses, cache memories are ineffective.

The data prefetching approach was thus proposed to reduce the processor stall time when applications lack temporal or spatial locality. As the name indicates, data prefetching is a technique to fetch data in advance. The essential idea is to observe data referencing patterns, then to speculate future references, and to fetch the predicted reference data closer to the processor before the processor demands them. Numerous studies have been conducted and many strategies have been proposed for data prefetching [2-5][8][10-17][19][23]. These studies concluded that prefetching is a promising solution to reducing access latency. The ultimate goal of data prefetching is to reduce access delay. However, the performance gain (how much we can reduce access delay) depends on many factors, such as prefetch coverage and accuracy. While computing

(c) 2007 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
SC07 November 10-16, 2007, Reno, Nevada, USA
(c) 2007 ACM 978-1-59593-764-3/07/0011…$5.00

capability is still increasing with a much faster pace than memory performance, more aggressive prefetching algorithms are desired, which provide wider coverage and higher accuracy. In the meantime, application features dominate referencing patterns. There is no single universal prefetching algorithm suitable for all applications. It is beneficial to support adaptive algorithms based on data access histories.

As the processor-memory performance gap increases, application features demand faster access to data, and hardware technologies evolve, we argue that it is time to dedicate one cache for prefetching to fully harvest benefits of aggressive, adaptive and other data prefetching strategies. We thus propose a dedicated prefetching cache structure, named Data Access History Cache (DAHC), and present data prefetching mechanisms to address this fundamental issue. The rest of this paper is organized as follows. Section 2 introduces the proposed DAHC design and methodology to serve multiple prefetching algorithms. Section 3 discusses our simulation experiments and performance results in detail to verify DAHC design and to demonstrate the potential performance improvement brought by DAHC-based data prefetching. Section 4 reviews related works and compares them with our approaches. Finally, we summarize our current work and discuss future work in Section 5.

2. DATA ACCESS HISTORY CACHE

The main purpose of the proposed DAHC is to track recent data access histories and maintain the correlations from different perspectives. Those histories and correlations are valuable information for data prefetching, especially for aggressive and adaptive strategies. In existing work, only very limited correlations are maintained, which limits the prefetching accuracy, coverage, and aggressiveness. Moreover, they only target a specific algorithm and have difficulty applying to diverse applications. However, with advances of processor technologies and the rapidly growing performance gap between processor unit and memory unit, it would be beneficial to trade computing power for a reduction in data access latency. With this idea, we propose to dedicate a cache (DAHC) for tracking data accesses and letting the processing unit perform comprehensive data prefetching. Therefore, processor stall time due to data accesses could be reduced and the overall system performance would be increased.

2.1 Design and Methodologies

The key idea of the DAHC is that history-based prefetching algorithms must rely on correlations within either program counter stream or data address stream, or both. Thus, the DAHC is designed to have three tables: one data access history table (DAH) and two index tables (PC index table and address index table). The DAH table accommodates history details, while the PC index table and the address index table maintain correlations from the PC and data address stream viewpoints respectively. A prefetching implementation can access these two tables to obtain the required correlations as necessary. Figure 1 illustrates the general design of DAHC and a high-level view of how it can be applied to support various prefetching algorithms.

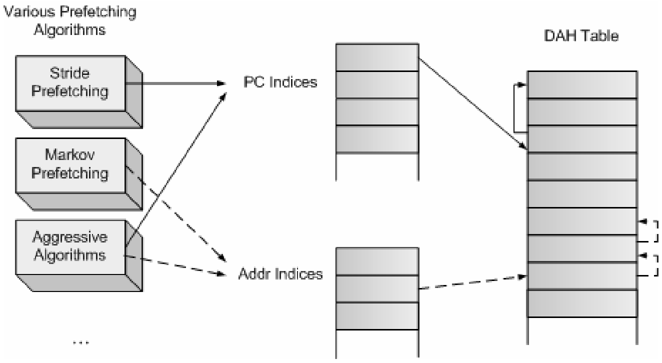
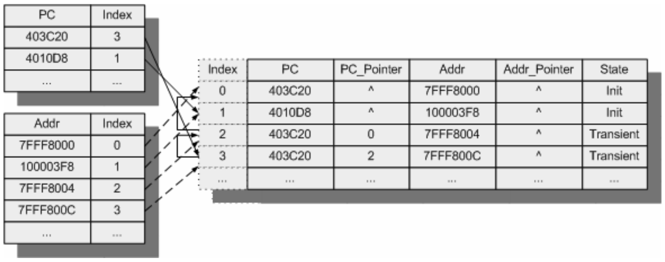


Figure 1. DAHC general design and high-level view

The detailed design of the DAHC is shown in Figure 2 through an example. The DAH table consists of PC, PC_Pointer, Addr, Addr_Pointer and State fields. PC and Addr fields store the instruction address and data address separately. The PC_Pointer and Addr_Pointer point to an entry where the last access from the same instruction or the last access of the same address is located. Therefore, PC_Pointer and Addr_Pointer link all accesses from the instruction stream and data stream perspectives. This design offers the fundamental mechanism to detect potential correlations and access patterns. The State field maintains state machine status used in prefetching algorithms. Various algorithms could occupy different bits of this field for maintaining their own states. The length of this field is implementation dependent, and the usage is decided by prefetching strategies.

The PC index table has two fields, PC and Index. The PC field represents the instruction address, which is a unique index in this table. The Index field records the entry of the latest data access in the DAH table from the instruction stored in the correspondent PC field. It is the connection between the PC index table and the DAH table. The address index table is similarly defined. For instance, in Figure 2, the DAH table captured four data accesses, three of them issued by instruction 403C20 (stored in the PC field) and one by instruction 4010D8. The instruction 403C20 accessed data at address 7FFF8000, 7FFF8004 and 7FFF800C in sequence, which is shown through the Addr and PC_Pointer fields. The instruction 403C20 and 4010D8 are also stored in the PC index table, and the corresponding Index field tracks the latest access from the DAH table, which are entry 3 and 1 respectively. The address index table keeps each accessed address and the latest entry, as shown in the bottom left of the figure, thus connecting all the data accesses on the basis of the address stream. Both PC index table and address index table can be implemented in a variety of ways including a fully associative structure and a set-associative structure. Notice that DAHC design is general and it does not imply any restriction to the system environment. It works in CMP or SMT environment, as well as in multiple applications environment.



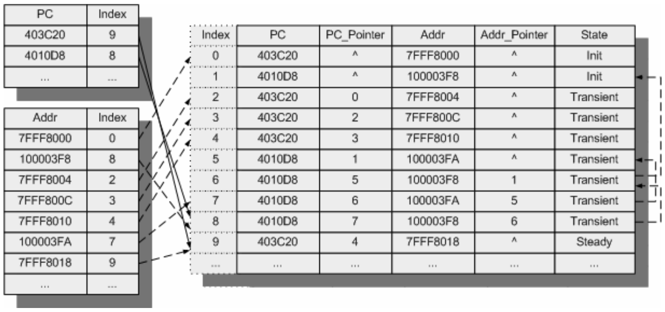
PC	Index
403C20	3
4010D8	1
...	...

Addr	Index
7FFF8000	0
100003F8	1
7FFF8004	2
7FFF800C	3
...	...

Index	PC	PC_Pointer	Addr	Addr_Pointer	State
0	403C20	^	7FFF8000	^	Init
1	4010D8	^	100003F8	^	Init
2	403C20	0	7FFF8004	^	Transient
3	403C20	2	7FFF800C	^	Transient
...	...	...	...	...	...

Figure 2. DAHC blueprint: PC index table, address index table and DAH table

Figure 3 shows a snapshot of the DAHC after capturing more data accesses. The PC index table, address index table and DAH table are updated. The latest access entries for instruction 403C20 and 4010D8 become index 9 and 8, respectively. The address accessed and the corresponding entry are updated in the address index table. In this case, a complex structured stride pattern of (4, 8, 4, 8) is detected for instruction 403C20 after examining address 7FFF8000, 7FFF8004, 7FFF800C, 7FFF8010 and 7FFF8018; therefore, data at address 7FFF801C and 7FFF8024 could be prefetched to memory in advance to avoid cache misses when 7FFF801C and 7FFF8024 are accessed as predicted. Such a complex structured pattern is a general case of stride pattern. However, the conventional stride prefetching approach [3] is unable to detect it without the DAHC support. This example also shows an address correlation between 100003F8 and 100003FA, which is often observed and utilized for prediction in the Markov prefetching algorithm [10] . The following section discusses data prefetching methodologies based on the proposed DAHC.



PC	Index
403C20	9
4010D8	8
...	...

Addr	Index
7FFF8000	0
100003F8	8
7FFF8004	2
7FFF800C	3
7FFF8010	4
100003FA	7
7FFF8018	9
...	...

Index	PC	PC_Pointer	Addr	Addr_Pointer	State
0	403C20	^	7FFF8000	^	Init
1	4010D8	^	100003F8	^	Init
2	403C20	0	7FFF8004	^	Transient
3	403C20	2	7FFF800C	^	Transient
4	403C20	3	7FFF8010	^	Transient
5	4010D8	1	100003FA	^	Transient
6	4010D8	5	100003F8	1	Transient
7	4010D8	6	100003FA	5	Transient
8	4010D8	7	100003F8	6	Transient
9	403C20	4	7FFF8018	^	Steady
...	...	...	...	...	...

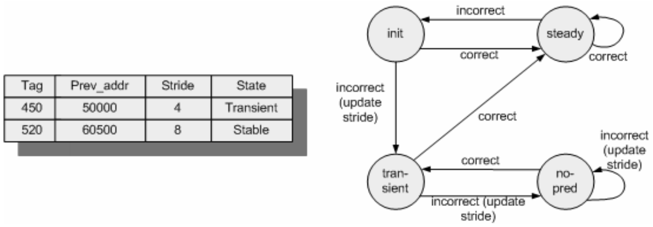
Figure 3. DAHC snapshot

2.2 DAHC-based Data Prefetching Mechanisms

2.2.1 Stride Prefetching

Stride prefetching predicts future references based on strides of recent references. This approach monitors data accesses and detects constant stride access patterns. Stride prefetching is usually implemented with a Reference Prediction Table (RPT) [3][7] as shown in Figure 4. RPT acts like a separate cache and holds data reference information of recent memory instructions. Since stride prefetching involves tracking the difference between two consecutive accesses and predicting the next access based on the stride, it is straightforward to design such an RPT table for stride prefetching implementation. Each entry in RPT is the instruction address, and it contains the last access address, the stride and the state transition information to predict future accesses. The right part of Figure 4 shows the state transitions. Once a pattern enters steady state or remains at steady state, which means a constant stride is found, a prefetch is triggered. The prefetched data address is simply calculated by adding the stride to the previous address.

Although RPT is effective for capturing constant stride of data accesses, it has several limitations. The first limitation is that RPT only calculates the stride between two consecutive accesses. It is hard to detect variable strides and impossible to find complex patterns, such as a repeating pattern of length *n* (e.g., 2, 4, 8, 2, 4, 8, …). Those complex patterns are common in user-defined data types. The second limitation is that RPT only tracks the last two accesses and omits many useful history references; thus, the accuracy in detecting patterns is relatively low. Those issues are addressed well in our proposed DAHC structure. Since DAHC tracks a large set of working histories, it is capable of detecting variable strides. Those detailed histories can also be used to improve the accuracy of stride detection. Moreover, DAHC makes detection of complex structure patterns possible, as discussed in previous examples.



Tag	Prev_addr	Stride	State
450	50000	4	Transient
520	60500	8	Stable

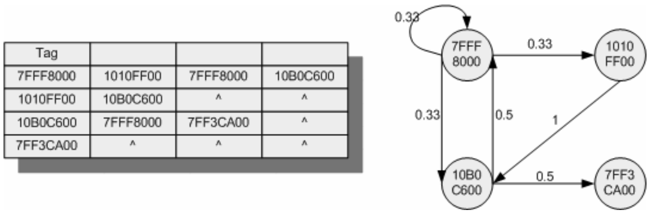
init
steady
tran-sient
no-pred
incorrect
correct
correct
incorrect (update stride)
correct
correct
incorrect (update stride)
incorrect (update stride)

Figure 4. Reference prediction table and state transition diagram

Stride prefetching can be implemented with the DAHC as follows. First, when a data access happens at monitoring level and is tracked by added DAHC component and related logic (see Section 3.1 for more details), the instruction address is searched for in the PC index table. If the instruction address does not match any entry in the PC index table, which means it is the first time that we see this instruction address in current working window, no prefetching action is triggered. If the instruction address matches one entry (it will match only one entry because the entries in index tables are unique), we follow the index pointer to traverse previous access addresses and detect whether a strided pattern or a structured pattern is present. If a pattern is detected, one or more data blocks are prefetched to data cache or a separate prefetch cache. The prefetching degree and prefetching distance can vary depending on the actual implementation. Finally, a new entry with this data access is created and inserted into the DAH table. The PC index table and address index table are updated correspondingly. Notice that the approach described above is enhanced stride prefetching with detection of variable and complex stride patterns. The conventional stride prefetching [3][7] can be implemented by detecting constant strides only.

2.2.2 Markov Prefetching

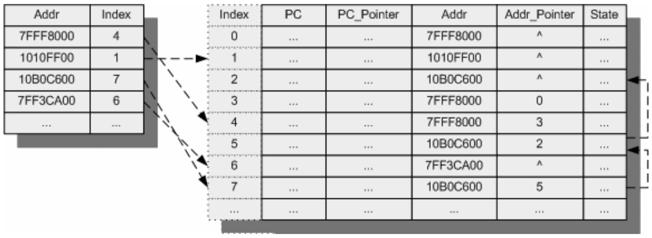
Markov prefetching is another classical prefetching strategy. The Markov prefetching algorithm builds a state transition diagram through past data accesses. The probability of each transition from one state to another state is calculated and updated dynamically. The algorithm assumes the future data accesses might repeat the histories. Therefore, once a new data access is captured, the future references predicted from the state transition diagram are prefetched in advance. For instance, Figure 5 shows the correlation table and state transition diagram for the data access stream 7FFF8000, 1010FF00, 10B0C600, 7FFF8000, 7FF3CA00, 7FFF8000, 10B0C600 and 7FF3CA00.



Tag			
7FFF8000	1010FF00	7FFF8000	10B0C600
1010FF00	10B0C600	^	^
10B0C600	7FFF8000	7FF3CA00	^
7FF3CA00	^	^	^

Figure 5. Markov prefetching correlation table and state transition diagram

The conventional Markov prefetching strategy treats all history accesses with the same weight. In practice, we usually give the highest weight to the latest access. This approach is essentially a combination of Markov model and LAST model [6]. The rationale is that the next data access is most probably the one that had followed the current access in the nearest past. For example, if we have a sequence of accesses to address A, B, A, C, D, A, then it is likely that the next access is C. With DAHC support, Markov prefetching can be implemented as follows. First, the data reference address is searched for within the address index table. If the newly accessed address does not match any existing entries, it is simply inserted into the DAH table. The PC index and address index table are also updated. If it matches an entry in the address index table, then we insert it to the DAH table and walk through the DAH table following the index and address pointer as shown in Figure 6. Each address next to these entries we visit is a prefetching candidate because each of this address was immediately accessed following the present access address in histories. Similar as in stride prefetching, different prefetching degree and prefetching distance can be supported depending on the actual implementation. If the prefetching degree is greater than one, we fetch multiple continuous data addresses following these entries we visit. We can also increase prefetching distance to initiate multiple visits. Continuing with the previous example and as shown in Figure 6, if a new data access address is 10B0C600, then a new entry is inserted into the DAH table at index 7, and the address index table is updated. After we walk through the DAH table following index 7, pointer 5 and pointer 2, data at address 7FF3CA00 and 7FFF8000 are prefetch candidates if we set prefetching degree as one and prefetching distance as two. Notice that Markov prefetching builds state transition based on data addresses. It does not need to use the state field.



Addr	Index
7FFF8000	4
1010FF00	1
10B0C600	7
7FF3CA00	6
...	...

Index	PC	PC_Pointer	Addr	Addr_Pointer	State
0	...	...	7FFF8000	^	...
1	...	...	1010FF00	^	...
2	...	...	10B0C600	^	...
3	...	...	7FFF8000	0	...
4	...	...	7FFF8000	3	...
5	...	...	10B0C600	2	...
6	...	...	7FF3CA00	^	...
7	...	...	10B0C600	5	...
...	...	...	...	...	...

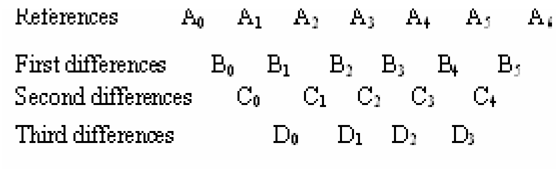
Figure 6. Markov prefetching with DAHC

2.2.3 Aggressive Prefetching Strategies

Since the DAHC maintains recent accesses in detail and the correlation among them, it is more powerful than supporting traditional prefetching approaches such as stride prefetching and Markov prefetching. It can support many other history-based prefetching strategies like more aggressive prefetching algorithms. It is an easy task to implement aggressive strategies with the DAHC because the DAHC is designed to support aggressive strategies naturally. The Multi-Level Difference Table (MLDT) prediction algorithm is such a representative aggressive strategy [21]. This prediction strategy forms a difference table of depth *d* of recent data accesses. Figure 7 demonstrates an example of the difference table. If a constant difference can be found in the first depth, which means a constant stride is found among data access histories, then the k^{th} future access from access A_r is predicted as $A_{r+k} = A_r + k * B$, where B is the constant difference among accesses. Some polynomial formula is used to predict the future access for general cases. For example, if a constant difference is found in the third depth, the future access is predicted as

$$A_{r+k} = A_r + k * B_{r-1} + \frac{k*(k+1)}{2} * C_{r-2} + M_k D .$$

Here $M_k = \frac{k}{6} * (k-1) * (k-2) + k^2$, where k = 1, 2…



References	A_0	A_1	A_2	A_3	A_4	A_5	A_6
First differences	B_0	B_1	B_2	B_3	B_4	B_5	
Second differences	C_0	C_1	C_2	C_3	C_4		
Third differences	D_0	D_1	D_2	D_3			

Figure 7. Example of difference table

MLDT strategy is similar to existing stride prefetching but is more aggressive since it searches references up to depth *d*. The stride prefetching is the special case where depth equals one. In addition, this method finds sets of repeating differences and ultimately finds the actual pattern in the accessing structures with variable stride data access patterns. For variable stride patterns, MLDT searches for regularity among data references by finding a deeper difference table. It can also be extended to find repeating sets of strides (e.g. 4, 8, 4, 4, 8, 4, 4, 8, 4…) at each level of difference table. Our proposed DAHC provides an implementation approach for the MLDT prefetching algorithm. First, when we see a data access at monitoring level, we check this access's instruction address with the PC index table. We update the DAH, PC index and address index tables as necessary. Second, we follow the index pointer and walk through the DAH table to find out previous accesses. These operations are similar as in stride prefetching case. The difference between MLDT prefetching and stride prefetching is that multiple level differences are calculated to detect if any constant stride, variable stride or complex structure pattern exists in each level, which means we perform a stride prefetching at each stride difference level. If a pattern is detected at some level, we stop going to further levels. If we continue to the further level, we calculate the strides of next level and they become the strides we deal with. Therefore, we always work with one level of stride similarly as in the conventional stride prefetching case. Figure 3 shows an example where a complex structure pattern (4, 8, 4, 8) is detected when we perform the MLDT prefetching with the DAHC.

2.3 Implementation Issues

The DAHC is straightforward and an effective prototype design of a prefetching-dedicated structure. It is a cache for data access information compared with conventional cache for instructions or data. The proposed DAHC can be placed at different levels for various desired data prefetching. For instance, it can be used to track all accesses to first level cache and to serve as a L1 cache prefetcher. It can also be placed at the second level cache and serves as a L2 cache prefetcher only. The straightforward design makes the implementation uncomplicated. The hardware implementation of the DAHC should be a specialized physical cache, like victim cache or trace cache. The PC index table and the address index table can be implemented with any associativity such as 2-way or 4-way. Since the index tables usually have less valid entries than the DAH table, it is unlikely that some entry is replaced due to a conflict miss. Even if a conflict miss occurs, it does not affect the correctness except discarding some access history. The DAH table can be implemented with a special structure where history information can be stored row by row and each row can be located by using its index. The logic to fill/update the DAHC comes from the cache controller. The cache controller traps data accesses at the monitored level and keeps a copy of the access information in the DAHC. If the DAH table is full, a victim entry will be selected and evicted out. The PC index table and the address index table are updated as well for consistency. The required DAHC size for normal applications' working set is trivial. For instance, if we suppose a DAHC with 1024 entries is implemented, which is a reasonable window size for a regular working set, then the required DAHC size is about 22KB. Our experiments simulated DAHC functionalities, and the conclusion is that DAHC is feasible in terms of hardware implementation.

3. SIMULATION AND PERFORMANCE ANALYSIS

We have conducted simulation experiments to study the feasibility of our proposed generic prefetching-dedicated cache, DAHC, for various prefetching strategies. Stride prefetching, Markov prefetching and MLDT aggressive prefetching algorithms were selected for simulation. This section discusses simulation details of DAHC-based data prefetching and presents the analysis results.

3.1 Simulation Methodology

The SimpleScalar simulator [1] was enhanced with data prefetching functionality to demonstrate how different prefetching algorithms can be implemented with the DAHC. The SimpleScalar tool set provides a detailed and high-performance simulation of modern processors. It takes binaries compiled for SimpleScalar architecture as input and simulates their execution on provided processor simulators. It has several different execution-driven processor simulators, ranging from extremely fast functional simulator to a detailed and out-of-order issue simulator, called the sim-outorder simulator.

We chose the sim-outorder simulator for our experiments. Figure 8 shows our modified SimpleScalar simulator architecture. We introduced two new modules: DAHC module and Prefetcher module. The DAHC module simulated the functionality of the proposed DAHC. Monitored data accesses were stored in the DAHC. The DAHC cache controller is responsible for updating all three tables. The Prefetcher module implemented the prefetching logic and different prefetching algorithms. In this module, a prefetch queue, similar to the ready queue of the original sim-outorder simulator, was created to store prefetch instructions. Prefetch instructions are similar to load instructions with a few exceptions. The first exception is that the effective address of each prefetch instruction is computed based on a data access pattern and prefetching strategy instead of computing the address using an integer-add functional unit. Another exception is that when prefetch instructions proceed through the pipeline, it is not necessary to walk through writeback and commit stages, and prefetch instructions do not cause any exceptions (prefetch instructions are silent). These similarities and differences provide us the guidelines to handle prefetch instructions. The implementation of prefetching strategies based on the DAHC follows the discussion given in Section 2.2.

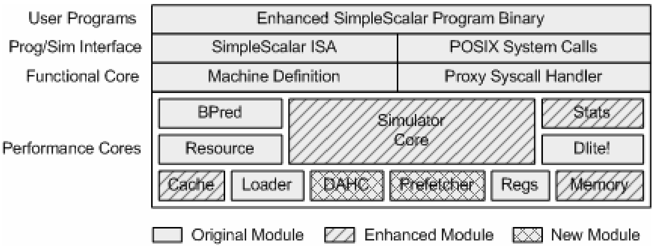


Figure 8. Enhanced SimpleScalar simulator

In addition to these two new modules, several existing modules were enhanced to incorporate the DAHC and data prefetching functionality. First, the simulator core module was revised to support the DAHC and Prefetcher modules. The pipeline was modified to have prefetching logic. The first improvement is each ready-to-issue load instruction is tracked to DAHC after the memory scheduler checks data dependencies. The prefetcher performs access pattern detection based on prefetching algorithms and makes prediction for future data accesses once a pattern is detected. Prefetch instructions are thus enqueued to prefetch queue. Another improvement is in instruction issue phase. During this phase, when we have available issue bandwidth, i.e. if there is idle bandwidth after issuing normal instructions, the prefetch queue is walked through and prefetch instructions are allocated with functional units to fetch the predicted data to data cache. Second, the memory module was modified to introduce a prefetch command to the memory component in addition to a load and a store command. The cache module was augmented with prefetch access handlers. Prefetch accesses can be handled similarly to load instructions except prefetch accesses do not cause any exceptions. Some additional statistics counters were added for measuring the effectiveness of prefetching.

Table 1. Simulator configuration

Issue width	4 way
Load store queue	64 entries
RUU size	256 entries
L1 D-cache	32KB, 2-way set associative, 64 byte line, 2 cycle hit time
L1 I-cache	32KB, 2-way set associative, 64 byte line, 1 cycle hit time
L2 Unified-cache	1MB, 4-way set associative, 64 byte line, 12 cycle hit time
Memory latency	120 cycles
DAHC	1024 entries
Prefetch queue	512 entries

3.2 Experimental Setup

We use the Alpha-ISA and configure the simulator as a 4-way issue and 256-entry RUU processor. The level one instruction cache and data cache are split. We configure L1 data cache as 32KB, 2-way with 64B cache line size. The latency is 2 cycles. L2 unified cache is configured as 1MB, 4-way with 64B cache line size. The latency of L2 cache is 12 CPU cycles. The DAHC is set as 1024 entries, and the replacement algorithm is FIFO. Both index tables are simulated with 4-way associative structures. We assume each DAHC access, such as a lookup within index tables, costs one CPU cycle. This should be a reasonable assumption for a small 4-way cache. We also assume a traversal within DAH table costs one cycle. If a prefetching algorithm needs to traverse multiple locations to make predictions, it consumes multiple cycles. The prefetch queue is set as 512 entries. Table 1 shows the configuration of our simulator.

3.3 Experimental Results

3.3.1 Matrix Multiplication Simulation

We first set up experiments to test the enhanced SimpleScalar simulator with DAHC-based data prefetching functionality. The prefetching strategy was set as the MLDT algorithm. Matrix multiplication was selected as the application because it is widely used in scientific computing and the correctness of its output results is easy to verify. The size of matrices was set as 200×200. We randomly generated the input, conducted simulation and then compared the output result with standard output to verify the correctness of the enhanced simulator. The correctness was also validated through checking the number of instructions (normal instructions) issued by the original and the enhanced version. The simulation results are shown in Table 2. The simulation time is the elapsed time for simulation (how much time the simulator spent in simulating). The results confirm that the enhanced SimpleScalar simulator worked correctly, and cache misses were reduced significantly through DAHC-based data prefetching.

Table 2. Simulation results for matrix multiplication

	# of instructions	Simulation Time	L1 cache misses	L1 replacements
Original	622140213	12633	1031047	1030023
Enhanced	622140213	13469	28772	1084326

3.3.2 SPEC CPU2000 Benchmark Simulation

We conducted several sets of SPEC CPU2000 benchmark [24] simulation for performance evaluation. Twenty-one of the total twenty-six benchmarks were tested successfully in our experiments. The other five benchmarks (apsi, facerec, fma3d, perlbmk and wupwise) had problems working under the SimpleScalar simulator (even in the original simulator) and did not finish the test.

The target of the first set of experiments was to compare the performance gain of traditional RPT-based stride prefetching approach and enhanced DAHC-based stride prefetching approach. Figure 9 shows the experimental results. The first bar in each test represents the level-one cache miss rate of the base case in which no prefetching was performed. The second and the third bar represent the miss rate in the case of RPT-based conventional stride prefetching and enhanced DAHC-based stride prefetching, respectively. As shown in Figure 9, the traditional approach reduced miss rates, and the enhanced approach reduced miss rates further. The rationale comes from that, with DAHC support, enhanced stride prefetching is able to detect complex structured patterns, and in addition, the prediction accuracy was improved through observing more histories. In contrast, many important and helpful histories were not considered and not fully utilized in traditional stride prefetching based on RPT.

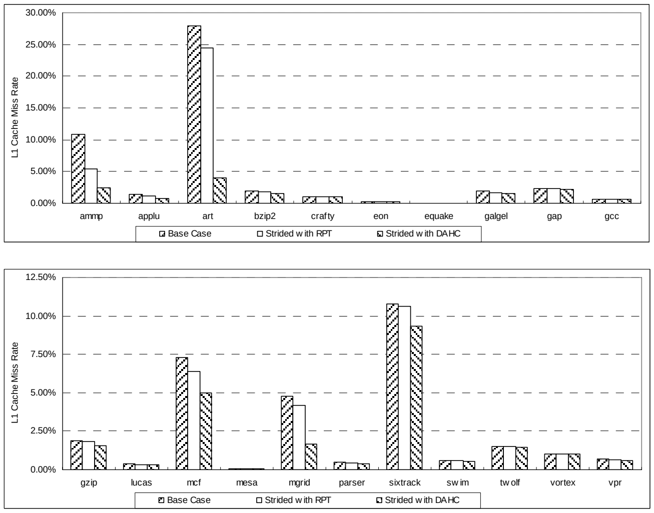


Figure 9. Stride prefetching with RPT vs. stride prefetching with DAHC

Figure 10 compares L1 cache miss rates of all tested SPEC CPU2000 benchmarks for the base case and three prefetching cases. This set of experiments showed that DAHC-based data prefetching worked well and the cache miss rates were reduced obviously in most cases. Among the three prefetching strategies, both stride and aggressive MLDT algorithms reduced a large ratio of miss rates. The MLDT algorithm was slightly better than stride prefetching because it searches more levels to find patterns among accesses. The Markov prefetching performed worse than stride and MLDT algorithms in most cases. One possible reason is that Markov prefetching requires a large set of states to characterize the probability of transition among accesses well. If the state diagram space is limited, it is hard for the Markov prefetching to guarantee the accuracy and coverage. Figure 11 illustrates L1 cache replacement rate in these tests. Cache pollution is considered a side effect of prefetching. An incorrect prediction brings a useless data block to cache and might replace useful data. With DAHC support, the prefetching accuracy increases by taking advantage of all available history information. As we can see from Figure 11, the replacement rate only increased slightly in DAHC-supported data prefetching.

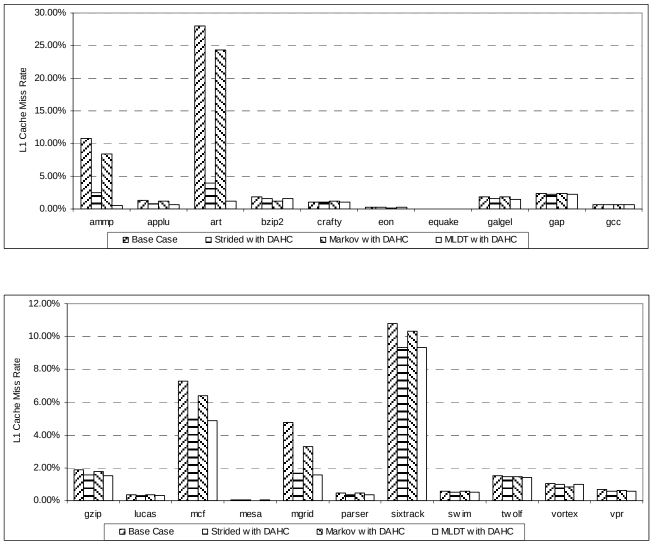


Figure 10. L1 cache miss rate of SPEC2000 benchmarks

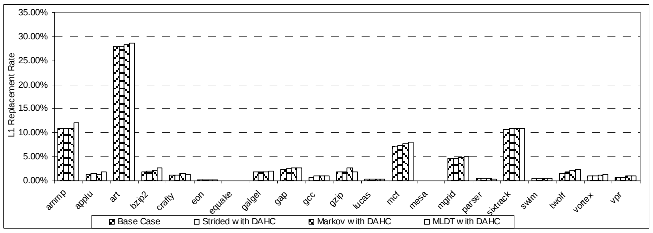


Figure 11. L1 cache replacement rate of SPEC CPU2000 benchmarks

Figure 12 shows the overall IPC (Instructions Per Cycle) improvement brought by three prefetching strategies: stride, Markov and MLDT prefetching based on DAHC. The experimental results demonstrated that the IPC value was improved considerably in most cases. The figure also reveals that even though MLDT achieved the best cache miss rate reduction in almost all cases, the IPC improvement was not always best. The stride prefetching outperformed the MLDT in the applu, crafty, gcc, gzip, lucas, mcf, parser, swim, twolf and vpr benchmarks. This is because MLDT involves more prefetching overhead for its aggressiveness due to more DAHC accesses. When we measured the overall system performance gain in IPC value, it paid for its additional overhead compared to stride prefetching. Another interesting fact shown in Figure 12 is that Markov strategy outperformed the other two in the bzip2, eon and vortex benchmarks. These facts confirmed that different strategies are desired for different applications to obtain the best prefetching benefits. It is necessary to support diverse algorithms and adapt to them dynamically based on distinct application features, and our proposed DAHC provides the essential structure support for adaptive strategies. Algorithm designers can utilize DAHC functionalities to come up with and implement adaptive algorithms.

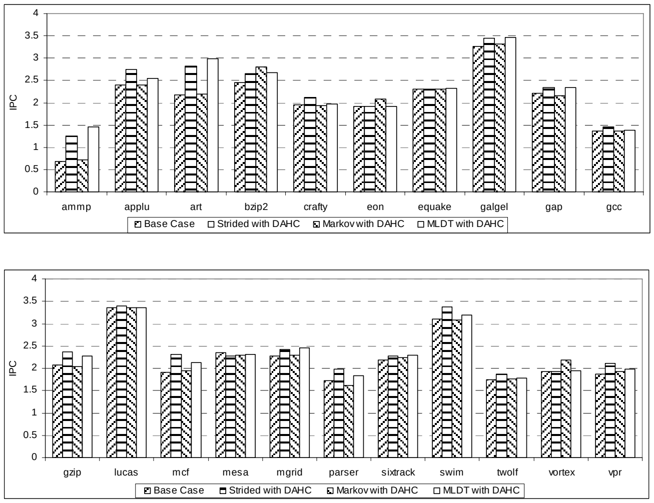


Figure 12. IPC value of SPEC CPU2000 benchmarks simulation

4. RELATED WORK

There are extensive research efforts in data prefetching area. Data prefetching is frequently classified as software prefetching and hardware prefetching [22]. Software prefetching instruments prefetch instructions to the source code either by a programmer or by a complier during the optimization phase. Recent work in helper threads [19], software-based speculative precomputation [12] [17] and data-driven multithreading [18] are such examples. The techniques include simple prefetching, unrolling the loop and software pipelining [22]. Software prefetching is usually used for large amount of loops. Such loops are very common in scientific computation, and these loops often exhibit poor cache utilization but have predictable memory-referencing patterns, and thus provide excellent prefetching opportunities.

Hardware-based prefetching does not require modifications to binary or source code and can benefit directly existing binary code. There is no need for programmer or compiler's intervention. Commonly used hardware prefetching techniques include sequential prefetching, stride prefetching and Markov prefetching. Sequential prefetching [4][5] fetches consecutive cache blocks by taking advantage of locality. The one-block-lookahead (OBL) approach automatically prefetches the next block when an access of a block is initiated. However, the limitation of this approach is that the prefetch may not be initiated early enough prior to processor's demand for the data to avoid a processor stall. To solve this issue, a variation of OBL prefetching, which fetches *k* blocks (called prefetching degree) instead of one block, is proposed. Another variation is called adaptive sequential prefetching, which varies prefetching degree *k* based on the prefetching efficiency. The prefetching efficiency is a metric defined to characterize a program's spatial locality at runtime. The stride prefetching approach [3] observes the pattern among strides of past accesses and thus predicts future accesses. Various strategies have been proposed based on stride prefetching, and these strategies maintain a reference prediction table (RPT) to keep track of recent data accesses. RPT provides a practical approach to implement stride prefetching, but the limitation is that only constant strides are recognizable. To capture repetitiveness in data reference addresses, Markov prefetching [10] was proposed. This strategy assumes the history might repeat itself among data accesses and build a state transition diagram with states denoting an accessed data block. The probability of each state transition is maintained so that the most probable predicted data are prefetched in advance and the least probable predicted data references can be dropped from prefetching. Other recent efforts in hardware prefetching include Zhou's dual-core execution (DCE) approach [23], Ganusov et al's future execution (FE) approach [8], Sun et al's data push server architecture [21] and Solihin et al.'s memory-side prefetching [20]. DCE and FE were proposed specifically for multi-core architecture. They use idle cores to pre-execute future loop iterations to warm up cache (bring data to cache in advance). The data push server architecture utilizes a separate processing unit such as a separate core to conduct heuristic prefetching. The memory-side prefetching approach uses a memory processor residing within main memory to observe data access histories and prefetch data proactively upon prediction. It is usually distinguished as push based prefetching from traditional pull based prefetching.

Without the benefit of programmer or compiler hints, the effectiveness of hardware prefetching largely relies on the accuracy of prediction strategies. Incorrect prediction brings useless blocks into cache, consumes memory bandwidth and might cause cache pollution. To increase prefetching accuracy and coverage, hardware prefetching strategies should be more aggressive. On the other hand, it is desired that data prefetching could support various algorithms and make dynamic selections because patterns are decided by application features and different prefetching algorithms are required for assorted applications. Our proposed generic and prefetching-dedicated DAHC cache was designed to resolve these issues. There are a few recent efforts in this area. Nesbit and Smith proposed a global history buffer for data prefetching in [14] and [15]. The similarity between their work and our work is that both attempt to facilitate data prefetching with a single structure. Their approach has demonstrated the feasibility of supporting different prefetching algorithms and achieved considerable performance gains. However, our work has substantial differences with theirs. First of all, we focus on providing a generic and dedicated cache for prefetching purposes and we argue that such a generic cache is a must to fully achieve prefetching benefits that hide access delay. Second, the global history buffer scheme is unable to support various algorithms simultaneously at runtime, and therefore, switching to different algorithms adaptively is impossible. Our work fully supports many history-based algorithms, as well as adaptive approaches, because we maintain two stream viewpoints concurrently. Third, we focus on supporting both algorithms' adaptability and aggressiveness. We believe that this strategy will help researchers fully utilize prefetching advantages. To our best knowledge, there is no other work targeting these directions. Another work closely related to this study is the instruction pointer based prefetcher developed by Intel [7]. The IP prefetcher is a RPT-like prefetcher; thus, it suffers the limitation that it only works for constant stride prefetching. Nevertheless, the Intel IP prefetcher provides us helpful guidelines in implementing the DAHC in hardware.

5. CONCLUSIONS AND FUTURE WORK

As memory performance lags far behind processor speed, data access delay has a severe impact on overall system performance. This study targeted to resolve this issue through fully exploiting data prefetching benefits with a generic and prefetching-dedicated cache. Our main contributions in this study include: 1) introducing a novel concept of a prefetching-dedicated cache considering both hardware technologies and application feature trends; 2) providing the design of a prefetching cache structure DAHC, and simulating its functionalities with an enhanced SimpleScalar simulator; and 3) presenting DAHC-associated data prefetching methodologies and demonstrating its support for prefetching algorithms with three representative examples, stride prefetching, Markov prefetching and an aggressive prefetching algorithm, MLDT algorithm. Our simulation experiments showed that the DAHC is feasible and that DAHC-based data prefetching achieved considerable cache miss rate reductions and IPC improvements.

We have demonstrated the power of the DAHC in supporting diverse prefetching algorithms in this study. In our future research, we plan to extend this work in various aspects. One of them is adapting to different prediction algorithms based on the data requirements of applications and making such decisions dynamically at runtime. We plan to define efficiency criteria for prefetching algorithms and to provide feedback for different algorithms and then to choose the best algorithm at runtime. Another of our future works will be to devise even more comprehensive prefetching strategies to further explore the DAHC's potentials.

6. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their helpful comments and the shepherd for providing detailed and valuable suggestions. This research was supported in part by National Science Foundation under NSF grant EIA-0224377, CNS0509118, and CCF-0621435. Fermi National Laboratory is operated by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the United States Department of Energy.

7. REFERENCES

[1] D.C. Burger, T.M. Austin and S. Bennett. Evaluating Future Microprocessors: the SimpleScalar Tool Set. University of Wisconsin-Madison Computer Sciences Technical Report 1308, July, 1996.

[2] J.B. Carter and et. al. Impulse: Building a Smarter Memory Controller. In Proc. of the 5th International Symposium on High Performance Computer Architecture, 1999.

[3] T.F. Chen and J.L. Baer. Effective Hardware-Based Data Prefetching for High Performance Processors. IEEE Trans. Computers, pp. 609-623, 1995.

[4] F. Dahlgren, M. Dubois, and P. Stenström. Fixed and Adaptive Sequential Prefetching in Shared-memory Multiprocessors. In Proc. 1993 International Conference on Parallel Processing, pp. I56-I63, 1993.

[5] F. Dahlgren, M. Dubois, and P. Stenström. Sequential Hardware Prefetching in Shared-Memory Multiprocessors. IEEE Trans. on Parallel and Distributed Systems, Volume 6, Issue 7, pp.733-746, 1995.

[6] P. Dinda, D. O'Hallaron. Host Load Prediction Using Linear Models. Cluster Computing, Volume 3, Number 4, 2000.

[7] J. Doweck. Inside Intel Core Microarchitecture and Smart Memory Access. Intel White Paper, 2006.

[8] I. Ganusov and M. Burtscher. Future Execution: A Hardware Prefetching Technique for Chip Multiprocessors. In Proc. of the 14th Annual International Conference on Parallel Architectures and Compilation Techniques, 2005.

[9] J. Hennessy and D. Patterson. Computer Architecture: A Quantitative Approach. The 4th edition, Morgan Kaufmann, 2006.

[10] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. In Proceedings of the 24th Annual Symposium on Computer Architecture, Denver-Colorado, pp 252-263, June 2-4 1997.

[11] A. C. Klaiber and H.M. Levy. An architecture for software-controlled data prefetching. SIGARCH Comput. Arch. News 19, 3 (May), 43-53, 1991.

[12] S. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. Shen. Post-Pass Binary Adaptation Tool for Software-Based Speculative Precomputation. In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’02), 2002.

[13] W.-F. Lin, S. K. Reinhardt and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In Proc. of the 7th International Symposium on High Performance Computer Architecture, pages 301.312, Jan 2001.

[14] K. J. Nesbit and J. E. Smith. Prefetching Using a Global History Buffer. In Proc. of the 10th Annual International Symposium on High Performance Computer Architecture (HPCA-10), Madrid, Spain, Feb. 2004: pages 96-106.

[15] K. J. Nesbit and J. E. Smith. Prefetching Using a Global History Buffer. IEEE Micro, 25(1), pp90-97, 2005.

[16] D.G. Perez, G. Mouchard and O. Temam. MicroLab: A Case for the Quantitative Comparison of Micro-Architecture mechanisms. In Proc. of the 37th International Symposium on Microarchitecture, 2004.

[17] M. Rodric and et. al. Compiler Orchestrated Pre-fetching via Speculation and Predication. In Proc. of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, 2004.

[18] A. Roth and G. S. Sohi. Speculative data-driven multithreading. In Proc. of the 7th International Symposium on High Performance Computer Architecture, 2001.

[19] Y. Song, S. Kalogeropulos and P. Tirumalai. Design and Implementation of A Compiler Framework for Helper Threading on Multi-Core Processors. In Proc. of 14th International Conference on Parallel Architectures and Compilation Techniques, 2005.

[20] Y.Solihin, J.Lee and J.Torrellas. Using a User-Level Memory Thread for Correlation Prefetching. In Proceedings of 8th International Symposium on Computer Architecture, 2002.

[21] X.H. Sun, S. Byna and Y. Chen. Improving Data Access Performance with Server Push Architecture. In Proc. of the NSF Next Generation Software Program Workshop in IPDPS’07, 2007.

[22] S. P. VanderWiel and D. J. Lilja. When caches aren't enough: Data prefetching techniques. IEEE Computer, 30(7):23--30, Jul 1997.

[23] H. Zhou. Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window. In Proc. of the 14th

International Conference on Parallel Architectures and Compilation Techniques, 2005.

[24] Standard Performance Evaluation Corporation, SPEC Benchmarks, http://www.spec.org/