International Journal of Advanced Research in Engineering Technology & Science
ISSN: 2349-2819
Email: editor@ijarets.org Volume-7, Issue-6 June- 2020 www.ijarets.org

A SURVEY ON DATA ACCESS HISTORY CACHE BASED DATA PREFETCHING MECHANISMS

Dr.Yee Yee Soe
Faculty of Computer System and Technologies,
University of Computer Studies,
Hpa-an, Kayin State, Myanmar (Burma)

ABSTRACT:
Data prefetching is an essential process to span the growing performance gap between processor and memory. While computing ability is increasing much faster than memory performance, it is time to have a allocated cache to store data access histories and to serve prefetching to hide data access latency effectively are proposed. A new cache structure, named Data Access History Cache (DAHC), is proposed and studied its related prefetching mechanisms. The DAHC acts as a cache for current reference facts instead of as a traditional cache for instructions or data. Theoretically, it is effective supporting numerous familiar history-based prefetching algorithms, mainly adaptive and aggressive approaches. The simulation experiments to prove DAHC design and DAHC-based data prefetching methodologies and to indicate performance gains are supported. The DAHC prepares a practical approach to getting data prefetching benefits and its associated prefetching techniques are showed more effective than traditional approaches.
Keywords: data prefetching, memory performance, data access latency, prefetching techniques, cache

INTRODUCTION:

While microprocessor performance improved by 52% a year until 2004 and has been increasing by 25%, memory speed is only increasing by roughly 9% each year [2]. The performance disparity between processor and memory remains expanding. Deeper memory hierarchies have initiated to span this gap [2]. Each memory level closer to the processor is smaller and faster than the following lower level. The reason 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).When applications absence locality by a working set size larger than the cache and/or non-contiguous memory accesses, cache memories are ineffective.

When applications absence temporal or spatial locality, the data prefetching approach is suggested to decrease the processor stall time. As the name specifies, data prefetching is a technique to fetch data in advance. The essential idea is to see data referencing patterns, then to speculate following references, and to fetch the predicted reference data closer to the processor before the processor requests them. These works are finished that prefetching is a promising solution to reducing access latency. The ultimate aim of data prefetching is to reduce access delay. However, the performance gain depends on many components, such as prefetch coverage and accuracy. While computing capability is quiet increasing with a much faster speed than memory performance, more aggressive prefetching algorithms are wanted, which provide wider coverage and higher accuracy. In the meantime, application features dominate referencing patterns. There is no single universal prefetching algorithm satisfactory for all applications. It is beneficial to help adaptive algorithms based on data access histories.

While the processor-memory performance space expands, application characteristics demand faster access to data, and hardware technologies develop, it is measure to give one cache for prefetching to completely harvest benefits of aggressive, adaptive and other data prefetching strategies have argued. A dedicated prefetching cache structure, named Data Access History Cache (DAHC), and

presents data prefetching mechanisms is proposed to address this fundamental issue. Section 2 introduces the suggested DAHC design and methodology to serve multiple prefetching algorithms. Section 3 discusses the simulation experiments and performance results in detail to prove DAHC design and to demonstrate the potential performance improvement proposed by DAHC-based data prefetching. Section 4 reviews related works and compares them with approaches. Finally, current work is summarized and discussed future work in Section 5.

DATA ACCESS HISTORY CACHE:

The main purpose of the proposed DAHC is to path recent data access histories and support the correlations from different perspectives. Those histories and connections are valuable information for data prefetching, especially for aggressive and adaptive strategies. In existing work, only very finite correlations are supported, which limits the prefetching accuracy, coverage, and aggressiveness. Moreover, they only target a particular algorithm and have difficulty applying to various applications. However, with advances of processor technologies and the rapidly growing performance gap between processor unit and memory unit, it can be advantageous to occupy computing power for a reduction in data access latency. With this plan, the dedicated a cache (DAHC) for tracking data accesses and allowing the processing unit perform comprehensive data prefetching are proposed. Therefore, processor stall time due to data accesses could be reduced and the general system performance could be increased.

Design and Methodologies

The key design of the DAHC is that history-based prefetching algorithms must depend on connections within either program counter stream or data address stream, or both. Therefore, the DAHC is arranged to possess three tables such as one data access history table (DAH) and two index tables (PC index table and address index table). The DAH table helps history features, while the PC index table and the address index table carry correlations from the PC and data address stream viewpoints respectively. A prefetching implementation can entry these two tables to get the required correlations as necessary. Figure 1 illustrates the common design of DAHC and a high-level perspective of how it can be applied to help different prefetching algorithms.

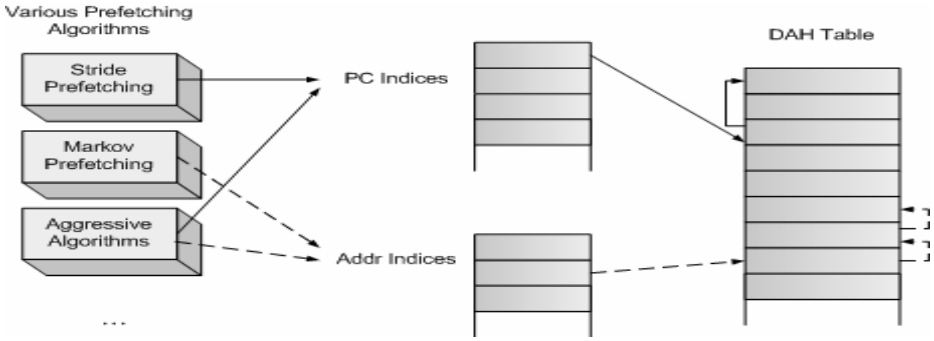


Figure 1. DAHC general design and high-level view

The comprehensive design of the DAHC is shown in “Figure 2” through an example. The DAH table contains PC, PC_Pointer, Addr, Addr_Pointer and State fields. PC and Addr fields supply the instruction address and data address separately. The PC_Pointer and Addr_Pointer point to an item where the last access from the same instruction or the last access of the same address is discovered. Therefore, PC_Pointer and Addr_Pointer relation are showed all accesses from the instruction stream and data stream views. This design provides the basic mechanism to detect potential correlations and access patterns. The State field keeps state machine status used in prefetching algorithms. The different algorithms can occupy different bits of this field for supporting their own states. The length of this field is dependently executed, and the usage is determined by prefetching strategies. The PC index table possesses two fields, PC and Index.

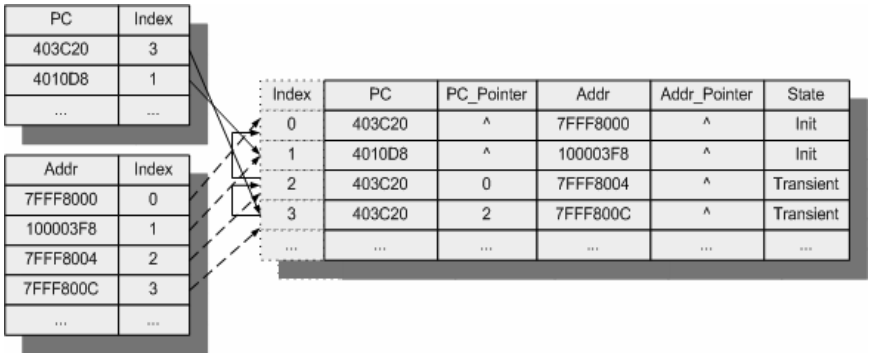


Figure 2. DAHC blueprint: PC index table, address index table and DAH table

The PC field constitutes 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 keeped in the equivalent PC field. This is the connection linking the PC index table and the DAH table. The address index table is similarly defined. The DAH table expressed four data accesses, three of them issued by instruction 403C20 (stored in the PC field) and one by instruction 4010D8 are shown in Figure 2. The instruction 403C20 retrieved data at address 7FFF8000, 7FFF8004 and 7FFF800C in sequence are showed in Addr and PC_Pointer fields. The instruction 403C20 and 4010D8 are also keeped 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 and associates all the data accesses on the basis of the address stream as shown in the bottom left of the "Figure 2". Both PC index table and address index table can be executed in a variety of methods including a fully associative structure and a set-associative structure. Notice that DAHC design is common and it does not involve any restriction to the system environment. It also tasks in CMP or SMT environment in multiple applications environment.

A snapshot of the DAHC after capturing more data accesses is showed in "Figure 3". The PC index table, address index table and DAH table are informed. The newest access entries for instruction 403C20 and 4010D8 are become index 9 and 8, individually. The address retrieved and the corresponding entry is updated in the address index table. In this case, a compound structured stride pattern of (4, 8, 4, 8) is noticed for instruction 403C20 after examining address 7FFF8000, 7FFF8004, 7FFF800C, 7FFF8010 and 7FFF8018. When 7FFF801C and 7FFF8024 are accessedas predicted, the data at address 7FFF801C and 7FFF8024 can be prefetched to memory in progress to avoid cache misses. Such a complex structured pattern is a common case of stride pattern. However, the normal stride prefetching approach [3] is unable to notice it without the DAHC support. This example shows an address connection between 100003F8 and 100003FA, which is noticed and utilized for prediction in the Markov prefetching algorithm [7]. The next section examines data prefetching methodologies based on the proposed DAHC.

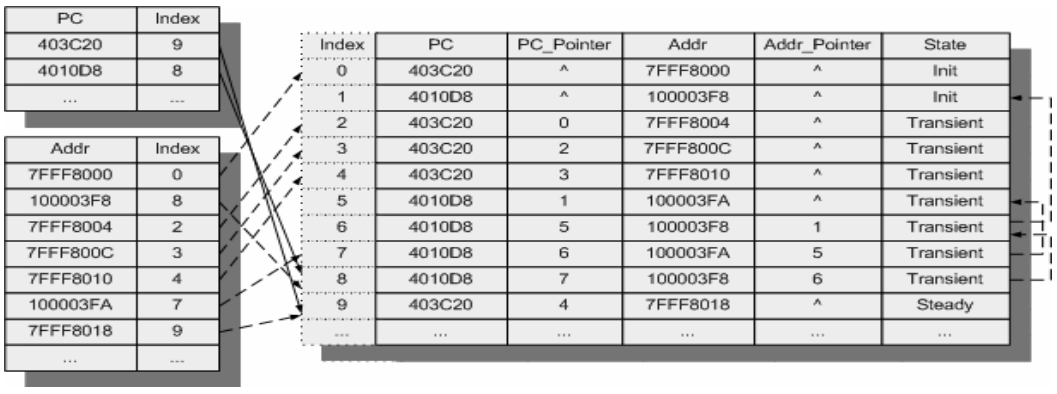


Figure 3. DAHC snapshot

DAHC Approached Data Prefetching Mechanisms:

Stride Prefetching

Stride prefetching predicts following references based on strides of recent references. This approach observes data accesses and detects constant stride access patterns. Stride prefetching is usually executed with a Reference Prediction Table (RPT) [5] as shown in "Figure 4". Reference Prediction Table behaves as a different cache and holds data reference facts of new memory instructions. Since stride prefetching includes following the contrast between two successive entries and forecasting the next access based on the stride, it is simple to arrangement such an RPT table for stride prefetching implementation. Every item in Reference Prediction Table (RPT) is the instruction address, and it carries the previous access address, the stride and the state transition information to foresee following accesses. The state transitions are exhibited in the right part of "Figure 4". When a sample enters steady state or remains at steady state, which indicates a constant stride is initiated, a prefetch is activated. The prefetched data address is directly computed by adding the stride to the previous address.

Although RPT is effective for expressing constant stride of data accesses, it has some restrictions. The first limitation is that RPT only computes the stride between two successive accesses. It is heavy to notice variable strides and impossible to find composite samples, such as a repeating pattern of length *n* (e.g., 2, 4, 8, 2, 4, 8, …). Those complex patterns are usual in user-defined data types. The second limitation is that RPT only tracks the last two accesses and forgets many useful history references. Thus, the accuracy in detecting patterns is relatively small. Those issues are addressed well in proposed DAHC structure. Since DAHC marks a large place of working histories, it is capable of detecting changeable strides. Those full histories can also be used to improve the accuracy of stride detection. Furthermore, DAHC builds detection of complex structure patterns possible.

Stride prefetching can be executed with the DAHC as follows. When a data access occurs at tracking level and is followed by attached DAHC part and related logic (see Section 3.1 for more feature), the instruction address is explored for in the PC index table. If the instruction address does not match any entry in the PC index table, which indicates it is the first time that this instruction address is seen in present 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), the index pointer to traverse previous access addresses is followed and discovered whether a strided pattern or a structured pattern is present. If a design is noticed, one or more data blocks are prefetched to data cache or a separate prefetch cache. The prefetching degree and prefetching distance can change depending on the real application. Finally, a new entry with this data access is generated and placed into the DAH table. The PC index table and address index table are modernized correspondingly. Note that the proposal related above is increased stride prefetching with observation of variable and compound step samples. The standard stride prefetching [2] can be applied by detecting constant strides only.

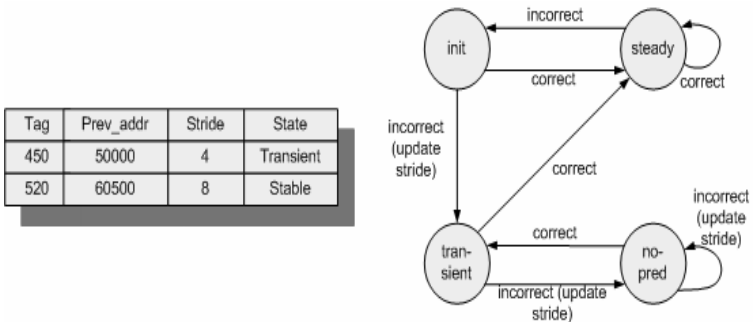
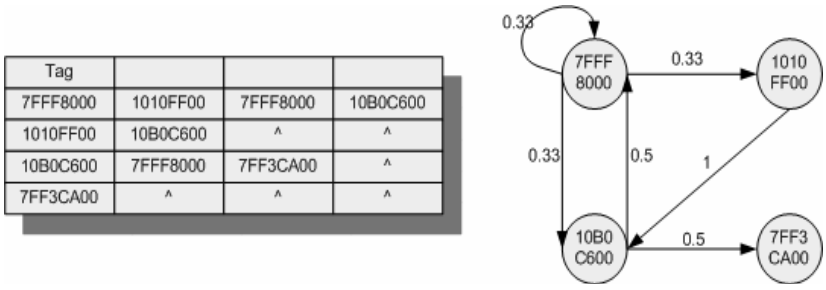


Figure 4. Reference prediction table and state transition diagram

Markov Prefetching

Markov prefetching is another traditional prefetching strategy. The Markov prefetching algorithm illustrates a state transition diagram. The Markov prefetching algorithm describes a state transition diagram through over data accesses. The probability of each transition from one state to another state is computed and updated dynamically. The algorithm supports the future data accesses might redo the histories. After a new data access is represented, the following references predicted from the state transition diagram are prefetched in proceed.For example, the correlation table and state transition diagram for the data access stream 7FFF8000, 1010FF00, 10B0C600, 7FFF8000, 7FF3CA00, 7FFF8000, 10B0C600 and 7FF3CA00 is shown in "Figure 5".

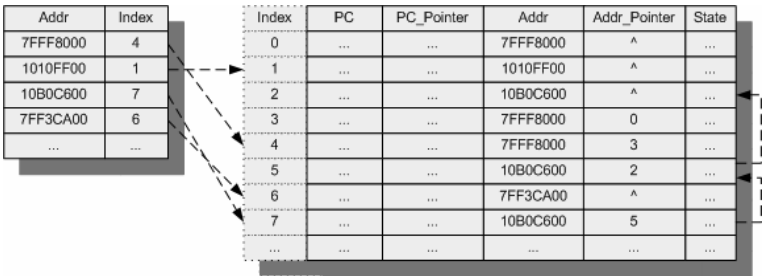


Tag			
7FFF8000	1010FF00	7FFF8000	10B0C600
1010FF00	10B0C600	^	^
10B0C600	7FFF8000	7FF3CA00	^
7FF3CA00	^	^	^

Figure 5. Markov prefetching correlation table and state transition diagram

The standard Markov prefetching strategy tends all history accesses with the same weight. In practice, the highest weight to the latest access is specified. This approach is necessary a mixture of Markov model and LAST model [3]. The reason is that the next data access is most probably the one that followedthe contemporary access in the nearest past. For example, if a sequence of accesses to address A, B, A, C, D, A has had, then it is likely that the next access is C. With DAHC support, Markov prefetching can be applied as follows. First, the data reference address is looked for within the address index table. If the recently accessed address does not match any existing entries, it is simply placed into the DAH table. The PC index and address index table are also modernized.

If it matches an entry in the address index table, then it is inserted 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 is a prefetching applicant because each of this address is immediately retrieved following the present access address in the past. The different prefetching level and prefetching interval can be helped depending on the actual implementation similar as in stride prefetching. If the prefetching degree is greater than one, multiple continuous can get and data addresses following these entries are overtaken. The prefetching distance is also grown to initiate multiple visits. If a new data access address is 10B0C600, then a new entry is put into the DAH table at index 7, and the address index table is updated in resuming with the foregoing example as shown in "Figure 6". After the DAH table following index 7, pointer 5 and pointer 2 are accompanied, statistics at address 7FF3CA00 and 7FFF8000 are prefetched candidates if prefetching stage as one and prefetching distance as two are set. The Markov prefetching builds state transition based on data addresses. It does not require to work 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

Aggressive Prefetching Strategies:

Since the DAHC supports new accesses in particular and the connection among them, it is more robust than aiding conventional prefetching approaches such as stride prefetching and Markov prefetching. It can carry many further history-based prefetching strategies like more aggressive prefetching algorithms. It is an easy job to execute aggressive strategies with the DAHC because the DAHC is plotted to support aggressive strategies naturally. The Multi-Level Difference Table (MLDT) prediction algorithm is a typical aggressive strategy [9]. This prediction strategy forms a difference table of depth *d* of new data accesses. Figure 7 indicates an example of the difference table. If a constant difference can be initiated in the first depth, which indicates a constant stride is started among data access reports, then the kthsubsequent access from access $A_r$is predicted as $A_{r+k}= A_r+ k\times B$, where Bis the constant difference among accesses. Some polynomial formula is used to forecast the later access for general cases. For example, if a constant difference is found in the third depth, the future access is predicted as equation (1).

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

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

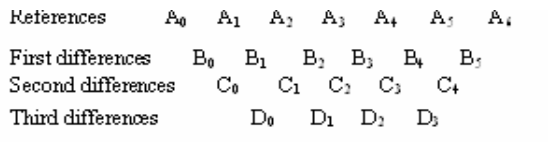


Figure 7. Example of difference table

MLDT strategy is near to existing stride prefetching but is further aggressive as it looks references up to depth d. The stride prefetching is the special occurrence where depth equals one. Additionally, this technique discovers sets of repeating differences and ultimately finds the true pattern in the accessing structures with variable stride data access patterns. For changeable stride patterns, MLDT looks for uniformly among data references by occuring a deeper difference table. It can also be expanded to find repeating sets of strides (e.g. 4, 8, 4, 4, 8, 4, 4, 8, 4…) at each quantity of difference table. DAHC gives an implementation approach for the MLDT prefetching algorithm. First, when a data access is seen at tracking level, this access's instruction address is examined with the PC index table. The DAH, PC index and address index tables are updated as necessary. Second, the index pointer is followed and walked through the DAH table to find out previous accesses. These workings are similar as in stride prefetching case. The difference between MLDT prefetching and stride prefetching is that multiple level differences are computed to notice if any constant stride, variable stride or complex structure pattern exists in each level, which conveys a stride prefetching at each stride difference level has performed. If a pattern is detected at some level, it is stopped going to further levels. If a pattern is continued to the further level, the strides of next level are calculated and they become the strides are dealed with. Therefore, it is always tasked with one level of stride similarly as in the normal stride prefetching case. When the MLDT prefetching with the DAHC is done, where a compound order fashion (4, 8, 4, 8) is detected as an example shown in "Figure 3".

Implementation Issues:

The DAHC is simple and a successful prototype design of a prefetching-dedicated structure.It is a cache for data access information compared with standard cache for instructions or data. The proposed DAHC can be placed at separate levels for different desired data prefetching. For example, it can be used to track all accesses to first level cache and to serve as a L1 cache prefetcher. It can also be set at the second level cache and serves as a L2 cache prefetcher only. The simple design makes the application uncomplicated. The hardware performance of the DAHC should be a specific physical cache, such as victim cache or trace cache. The PC index table and the address index table can be executed with any connectivity such as 2-way or 4-way. Since the index

tables usually have less correct entries than the DAH table, it is unlikely that some entry is replaced due to a conflict miss. Even if a disagreement miss occurs, it does not affect the correctness except discarding some access history. The DAH table can be applied 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 occurs the cache controller. The cache controller catches data accesses at the detected quantity and remains a copy of the entry information in the DAHC. If the DAH table is full, a victim entry will be selected and ejected out. The PC index table and the address index table are also updated for consistency. The needed DAHC size for normal applications' operating set is not important. These experiments are simulated DAHC functionalities and the conclusion is that DAHC is achievable in terms of hardware implementation.

SIMULATION AND PERFORMANCEANALYSIS:
The simulation experiments to study the feasibility of the proposed generic prefetching-dedicated cache are conducted DAHC for various prefetching strategies. Stride prefetching, Markov prefetching and Multi-Level Difference Table aggressive prefetching algorithms are chosen for simulation. This section surveys simulation characteristics of DAHC based data prefetching and introduces the analysis results.

Simulation Methodology:
The SimpleScalar simulator [1] is incorporated with data prefetching functionality to show how separate prefetching algorithms can be executed with the DAHC. The Simple Scalar tool set gives a detailed and high-performance simulation of modern processors. It gains binaries assembled for SimpleScalar construction as input and reproduces their execution on supplied processor simulators. It has some different execution-driven processor simulators, varying from extremely fast functional simulator to a detailed and out-of-order issue simulator, called the sim-outorder simulator.

The sim-outorder simulator is selected for experiments. Figure 8 shows modified SimpleScalar simulator architecture. This is initiated two new modules: DAHC module and Prefetcher module.The DAHC module is simulated the functionality of the proposed DAHC. Monitored data accesses are keeped in the DAHC. The DAHC cache controller is responsible for renovating all three tables. The Prefetcher module is applied the prefetching logic and different prefetching algorithms.

In this module, a prefetch queue, near to the prepare queue of the actual sim-outorder simulator, is produced to keep prefetch instructions.Prefetch instructions are similar to load instructions with a few irregularity. The first exception is that the essential address of each prefetch instruction is calculated based on a data entry pattern and prefetching strategy instead of determining the address using an integer-add functional unit. Another case is that when prefetch instructions go through the pipeline, it is not necessary to walk through writeback and do stages, and prefetch instructions do not cause any exceptions (prefetch instructions are silent). These similarities and differences allow the guidelines to handle prefetch instructions. The application of prefetching strategies based on the DAHC follows the discussion in part 2.2. Moreover, these two new modules, some existing modules are increased to include the DAHC and data prefetching functionality.

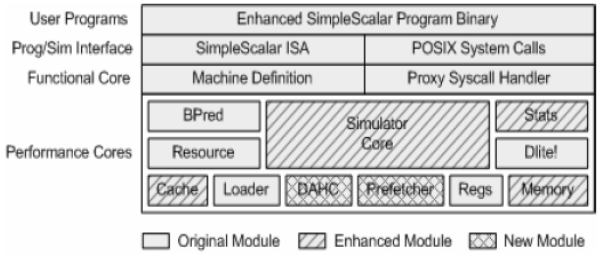


Figure 8. Enhanced SimpleScalar simulator

First, the simulator core module is reviewed to support the DAHC and Prefetcher modules. The pipeline is modified to experience prefetching logic. The first improvement is each ready-to-issue load instruction is pathed to DAHC after the memory scheduler examines data dependencies. The prefetcher performs access pattern detection based on prefetching algorithms and builds prediction for future data accesses once a pattern is detected. Prefetch instructions are enqueued to prefetch queue. Another improvement is in instruction issue phase. During this phase, when available issue bandwidth has, i.e. if there is idle bandwidth after issuing common instructions, the prefetch queue is walked through and prefetch instructions are assigned with functional units to fetch the predicted data to data cache. Second, the memory module is modified to begin a prefetch command to the memory component in addition to a load and a store command. The cache module is increased with prefetch access handlers. Prefetch accesses can be holded similarly to load instructions except prefetch accesses do not cause any exceptions. Some further statistics counters are added for measuring the effectiveness of prefetching.

Experimental Setup:

The Alpha-ISA has used and configured the simulator as a 4-way issue and 256-entry RUU processor. The level one instruction cache and data cache has split. L1 data cache as 32KB and 2-ways with 64B cache line size have configured. The latency is 2 cycles. L2 unified cache has configured as 1MB, 4-way with 64B cache line size. The latency of L2 cache has 12 CPU cycles. The DAHC has placed as 1024 entries, and the replacement algorithm is FIFO. Both index tables are simulated with 4-ways linked structures. Each DAHC access, such as a lookup within index tables, costs one CPU cycle have assumed. This has been a reasonable acceptance for a small 4-way cache. A traversal within DAH table costs one cycle is also assumed. If a prefetching algorithm requires crossing many sites to force predictions, it absorbs multiple patterns. The prefetch queue is set as 512 entries. “Table 1” shows the configuration of the simulator.

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

Experimental Results:

Matrix Multiplication Simulation

The demonstrations to experiment the enhanced SimpleScalar simulator with DAHC based data prefetching functionality are arranged. The prefetching strategy is set as the MLDT algorithm. Matrix multiplication is chosen as the application because it is widely used in scientific computing and the correctness of its output results is easy to confirm. The size of matrices is set as 200× 200. These results are 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 is also showed through checking the number of instructions (normal instructions) supplied by the original and the enhanced version. The simulation time is the passed time for simulation (how much time the simulator spent in simulating). The consequence confirm that the enhanced SimpleScalar simulator operated accurately, and cache misses are decreased 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

SPEC CPU2000 Benchmark Simulation:

The several sets of SPEC CPU2000 benchmark [8] simulation are conducted for performance evaluation. Twenty-one of the total twenty-six benchmarks are successfully tested in experiments. The more five standards (apsi, facerec, fma3d, perlbmk and wupwise) had difficulties operating down the SimpleScalar simulator (even in the original simulator) and had not concluded the experiment.

The select of the original position of tests had to differentiate the execution gain of conventional RPT-based stride prefetching approach and enhanced DAHC-based stride prefetching approach. Figure 9 shows the experimental results. The initial bar in each experiment means the level-one cache miss rate of the bottom instance in which no prefetching is executed. The second and the third stand represent the miss rate in the case of conventional stride prefetching with RPT and enhanced stride prefetching with DAHC individually.The conventional approach reduced miss rates, and the enhanced approach reduced miss rates further are shown in "Figure 9". The rationale comes from that, with DAHC support, enhanced stride prefetching is able to detect complex structured patterns, and then the prediction accuracy is improved through observing more histories. In contrast, numerous key and helpful records are not observed and not fully utilized in traditional stride prefetching based on RPT.

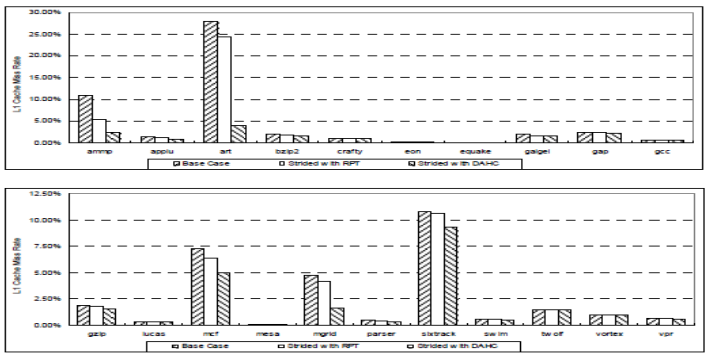


Figure 9. Stride prefetching with RPT vs. stride prefetching with DAHC

The comparison L1 cache miss rates of all tested SPEC CPU2000 benchmarks are shown in "Figure 10" for the base case and three prefetching cases. This set of experiments are exhibited that DAHC-based data prefetching worked completely and the cache miss rates are decreased clearly in most instances. Both stride and aggressive MLDT algorithms reduced a large ratio of miss rates among the three prefetching strategies. The MLDT algorithm had slightly better than stride prefetching because it explores more levels to discover patterns among accesses. The Markov prefetching is performed poor than stride and MLDT algorithms in most cases. One possible cause is that Markov prefetching needs a large set of conditions to distinguish the expectation of conversion among accesses effectively. If the state diagram space is restricted, it is smart for the Markov prefetching to warranty the accuracy and coverage.

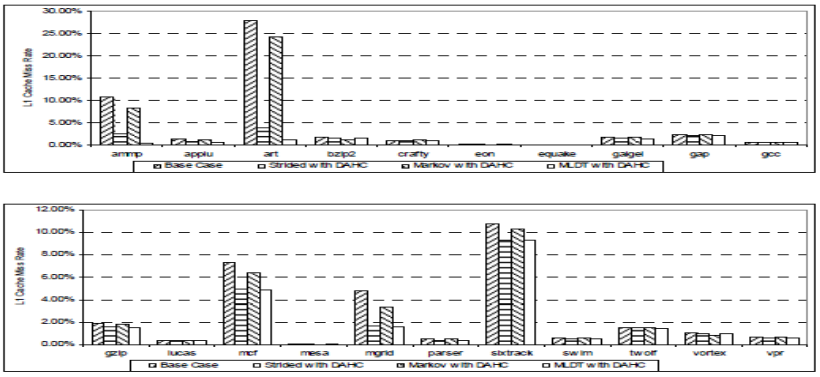


Figure 10. L1 cache miss rate of SPEC 2000 benchmarks

The L1 cache replacement charge in these experiments is shown in "Figure 11". Cache contamination is considered a side effect of prefetching. An incorrect prediction refers a useless data block to cache and might replace useful data. With DAHC support, the prefetching accuracy grows by taking advantage of all available history information. The replacement rate only expanded moderately in DAHC-supported data prefetching is seen in "Figure 11".

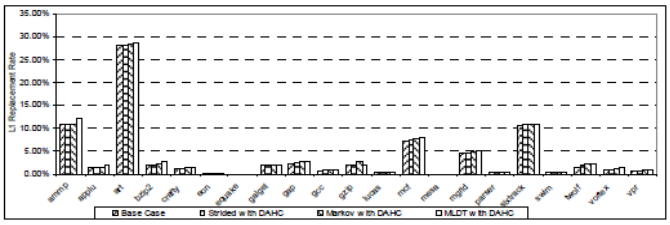


Figure 11. L1 cache replacement rate of SPEC CPU2000 benchmarks

The overall IPC (Instructions per Cycle) improvement preferred by three prefetching strategies: stride, Markov and MLDT prefetching based on DAHC are shown in "Figure 12". The experimental results demonstrated that the IPC values are improved greatly in most cases. The figure also reveals that even though MLDT attained the best cache miss rate depletion in nearly all instances, the IPC improvement is not always best. The MLDT in the applu, crafty, gcc, gzip, lucas, mcf, parser, swim, twolf and vpr benchmarks are outperformed the stride prefetching. This is because MLDT requires more prefetching overhead for its aggressiveness due to more DAHC accesses. When the general system performance benefit is considered in IPC value, it paid for its further above compared to stride prefetching.The Markov strategy outperformed the more two in the bzip2, eon and vortex benchmarks are shown in "Figure 12" as in another interesting reality. These facts established that individual strategies are wanted for different applications to get the best prefetching benefits. It is certain to help diverse algorithms and adapt to them dynamically based on clear application characteristics, and the proposed DAHC provides the key structure support for adaptive strategies. Algorithm creaters can use DAHC functionalities to come up with and implement adaptive algorithms.

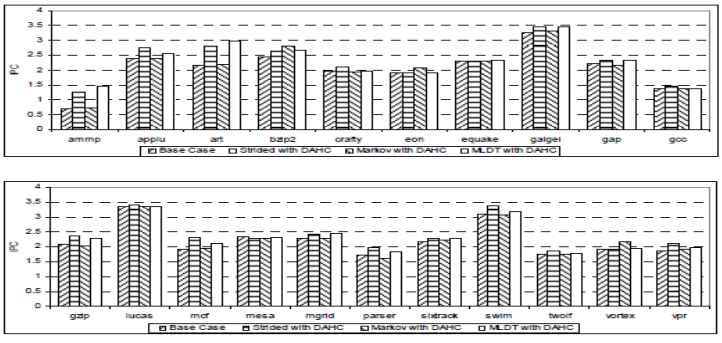


Figure 12. IPC value of SPEC CPU2000 benchmarks simulation

RELATED WORK:

Data prefetching is often classified as software prefetching and hardware prefetching. Software prefetching tools prefetch instructions to the source code either by a programmer or by a complier during the optimization period.

Hardware-based prefetching does not need modifications to binary or source code and can directly benefit living binary code. There is no require for programmer or compiler's intervention. The hardware prefetching approaches include sequential prefetching, stride prefetching and Markov prefetching. Sequential prefetching [6] fetches successive cache blocks by proceeding advantage of locality. The one-block-lookahead (OBL) approach automatically prefetches the following block when an entry of a block is initiated. Although, the restriction of this approach is that the prefetch may not be begun enough prior to processor's request for the data to avoid a processor stall. To solve this supply, a difference of OBL prefetching, which fetches *k* blocks (called prefetching degree) instead of one block, is suggested. Another form is called adaptive sequential prefetching, which differs prefetching degree *k* based on the prefetching efficiency. The prefetching efficiency is a metric determined to characterize a program's spatial locality at runtime. The stride prefetching approach [3] notices the pattern among strides of past accesses and predicts future accesses.

The different strategies are suggested based on stride prefetching and these strategies support a reference prediction table (RPT) to support path of current data accesses. RPT provides a actual approach to implement stride prefetching, but the limitation is that only constant strides are noticeable. To express repetitiveness in data reference addresses, Markov prefetching [7] is proposed. This strategy supposes 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 keeped so that the most probable predicted data are prefetched in advance and the least probable predicted data references can be dropped from prefetching.

The data push server architecture utilizes unconnected processing unit such as a separate core to behaviour heuristic prefetching. The memory-side prefetching approach works a memory processor occupying within main memory to see data access histories and prefetch data proactively upon prediction.It is usually differentiated as push based prefetching from traditional pull based prefetching.

The effectiveness of hardware prefetching mostly depends on the accuracy of prediction strategies by the absence of the interest of programmer or compiler hints. Incorrect prediction prefers useless blocks into cache, consumes memory bandwidth and may cause cache pollution. The hardware prefetching strategies should be more aggressive to increase prefetching accuracy and coverage.On the other hand, it is wanted that data prefetching could help various algorithms and construct dynamic selections because patterns are determined by application attribute and different prefetching algorithms are required for mixed applications. The proposed generic and prefetching-dedicated DAHC cache is designed to decide these issues.

The IP prefetcher is a RPT-like prefetcher. Thus, it suffers the constraint that it only performs for continuous stride prefetching. Nevertheless, the Intel IP prefetcher supplies the helpful guidelines in implementing the DAHC in hardware.

CONCLUSIONS AND FUTURE WORK:

The data access delay is a severe impact on overall system performance because of memory performance lags far behind processor speed. This study is targeted to determine this issue through completely exploiting data prefetching benefits with a generic and prefetching-dedicated cache. The main contributions in this learning are includedthe introducing a novel concept of a prefetching-dedicated cache considering both hardware technologies and application feature trends, providing the design of a prefetching cache structure DAHC and simulating its functionalities with

an enhanced SimpleScalar simulator and presenting DAHC-associated data prefetching methodologies and demonstrating its support for prefetching algorithms with three representative samples, stride prefetching, Markov prefetching and an aggressive prefetching algorithm, MLDT algorithm. The simulation experiments showed that the DAHC is feasible and that DAHC-based data prefetching achieved considerable cache miss rate reductions and IPC improvements.

The power of the DAHC in supporting various prefetching algorithms is demonstrated in this study. In future work, the extend of this work has planned in various aspects. One of them has modified to different prediction algorithms based on the data requirements of applications and maked such decisions dynamically at runtime. The efficiency criteria for prefetching algorithms has defined and provided feedback for different algorithms and then to choose the best algorithm at runtime. Another future works will be to devise even more comprehensive prefetching strategies to further explore the DAHC's potentials.

REFERENCES

1. **D.C. Burger, T.M. Austin and S. Bennett.**Evaluating Future Microprocessors: the SimpleScalar Tool Set. UniversityofWisconsin-Madison Computer Sciences Technical Report 1308, July, 1996.
2. **J. Hennessy and D. Patterson.** Computer Architecture: A Quantitative Approach. The 4th edition, Morgan Kaufmann, 2006.
3. **T.F. Chen and J.L. Baer.**Effective Hardware-BasedData Prefetching for High Performance Processors.IEEETrans.Computers, pp. 609-623, 1995.
4. **J. Doweck.**Inside Intel Core Microarchitecture and Smart Memory Access.Intel White Paper, 2006.
5. **P. Dinda, D. O'Hallaron.** Host Load Prediction Using Linear Models. Cluster Computing, Volume 3, Number 4, 2000.
6. **F. Dahlgren, M. Dubois, and P. Stenström.**Sequential Hardware Prefetching in Shared-MemoryMultiprocessors.IEEE Trans.onParallelandDistributedSystems,Volume6,Issue 7, pp.733-746, 1995.
7. **D. Joseph and D. Grunwald.**Prefetching Using Markov Predictors.In Proceedings of the 25th Annual Symposium on Computer Architecture, Denver-Colorado, pp 252-263, June 2-4 1998.
8. Standard Performance Evaluation Corporation, SPEC Benchmarks, http://www.spec.org/
9. **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 Workshopin IPDPS'07, 2007.