# 1991: An effective on-chip preloading scheme to reduce data access penalty

----1991 Jean-Loup Baer

With this scheme, data with regular access patterns is preloaded, independently of the stride size, a preloading of data with irregular access patterns is prevented.

**1 introduction**

Some enhanced architectural features to improve instruction execution rates: superscalar, vector data handling, cache hierarchies, multiprocessing.

In this paper, we propose the design of an on-chip hardware support function whose goal is to reduce the memory latency due to data cache misses, by preloading blocks before they are needed.

Prefetching problems: preload the wrong block; fail to preload in time; displace a useful block

Basic structure of our prefetch hardware: LA-PC + RPT

**2 Background and previous work**

* Hardware-based prefetching

OBL, prefetch block i+1 unconditionally, only on a miss to block I, or if the prefetch has been successful in the past

1982：cache memory, A. J. Smith

Argue against complex prefetch strategies:

1990：The performance impact of block sizes and fetch strategies

Write buffer：allow delaying the writes in favor of more urgent cache loads.

Stream buffer：are FIFO queues that are filled sequentially starting from the missing block address

1990：Improving directed-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers, N. P. Jouppi

Lock-free cache：lockup-free operation permits the initiation of cache loads for uncahed blocks that will be referenced in subsequent instructions while the fetching of some block is in progress

1981：lock-up free instruction fetch/prefetch cache organization, D. Kroft

Prefetching based on instruction stream execution

1987：Aspects of Cache Memory and Instruction Buffer Performance, M. D. Hill

Multiprocessor running data prefecth scheme

1987：Data Prefetching in shared memory multiprocessor, R. L. Lee

Implicit prefetching is present in decoupled architecture: two instruction streams operate concurrently, communicate via queues, and drive two execution units, one for data access and one for functional operations. The data access stream can be “ahead” of the functional stream and hence prefetch operands needed in the near future.

1982：Decoupled access/execute computer architecture, J. E. Smith

* Compiler-directed prefetching

Non-blocking cache load instruction: this instruction should be positioned enough in advance of the actual use of its operand

1989：Software methods for improvement of cache performance on supercomputer application, A. K. Porterfield

Finding the earliest time at which prefetching can be performed(after 1989 paper)

1990：Compiler-directed Data Prefetching in Multiprocessor with Memory Hierarchies, E. Gornish

**3 Data Cache Preloading**

Prefetching based on sequentiality can be successful for optimization of I-cache, but much less so for D-Cache. Therefore, we turn our attention solely to the case of D-Cache. The basis of our hardware-based scheme is to predict the instruction execution stream and the data access patterns far enough in advance, so that the required data can be preloaded and be in the cache when the real memory access instruction is executed.

**3.1 Motivation**

* Four categories of memory access patterns

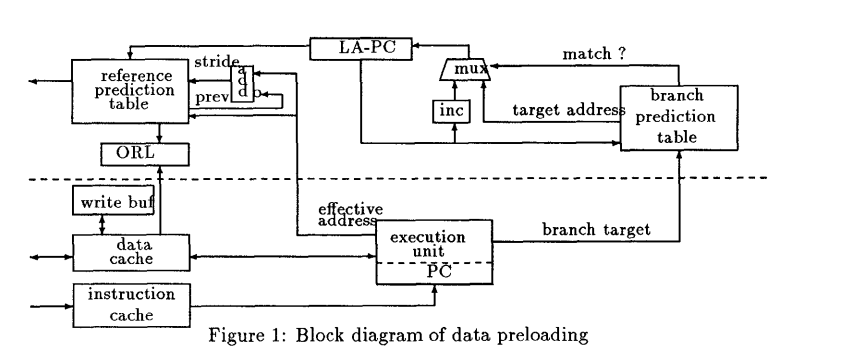
Scalar: simple variable reference

Zero stride: subscripts expression unchanged

Constant stride: subscript expression linear

Irregular: none of above

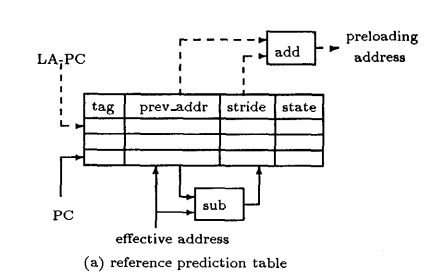
* Caches work well for scalar and zero stride references. Caches with large block size and most of the prefetch strategies discussed previously can improve the performance for the constant stride category if the stride is small but will be of no help if the stride is large.
* Data access patterns of load/store instructions will be kept in a RPT(Reference Prediction Table) which will be accessed ahead of time by LA-PC(Look-ahead PC). The look-ahead PC will be incremented as a regular PC and modified appropriately with the help of a BPT(Branch Prediction Table).

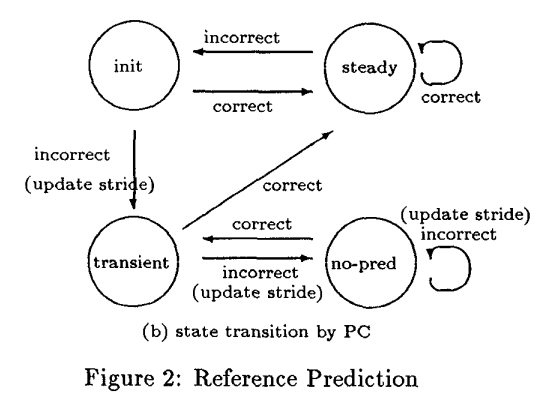
**3.2 Basic Block Diagram**

BPT(Branch Prediction Table)+LA-PC(Look-ahead PC)+RPT(Reference Prediction Table)+ORL(Outstanding Request List)

When the LA-PC hits a load/store instruction that already been stored in the RPT, a check is made to see whether (1)the state of the entry is for no prediction, or(2)the data is already in the cache, or(3)the preloading of the block is in progress(by checking ORL). If none of these three conditions is true, a request to load the operand is performed and its address is stored in ORL.

**3.3 Reference Prediction Table-RPT**

****

****

* LA-PC operation

A1: NO-operation: there is no corresponding entry in the RPT, or there is an entry in state no prediction;

A2: Potential preload: there is a corresponding entry, a block address (prev\_addr+stride) is generated. If the block is uncached and the address is not found in the ORL, a preload is initiated.

* PC operation

When the PC encounters a load/store instruction with effective operand address, RPT is update as follows

B1: there is no corresponding entry. The instruction is entered in RPT, the prev\_addr field is set to addr, the stride to 0, and the state to initial.

B2: there is a corresponding entry, then operation as status figure according correct or incorrect.

**3.4 Look Ahead Distance(LA-distance)**

Ideally, we would like to keep a Look Ahead distance between the current PC and LA-PC equal to the latency of the next level in the memory hierarchy.

**3.5 cache misses**

On a cache read miss, the cache controller checks the ORL. If the block has already been requested, a normal stall occurs. Otherwise, a regular load is issued with priority over the buffered preload requests. When the LA-PC has to be reset because of an incorrect branch prediction, the buffered preload requests are flushed. When a preload raises an exception (e.g., page fault) we ignore the preload.

**4 Methodology**

**4.1 Trace-driven simulation**

* Incorrect reference predictions could be caused by :

incorrect branch predictions,;

RPT entry conflicts;

incorrect states in the RPT table.

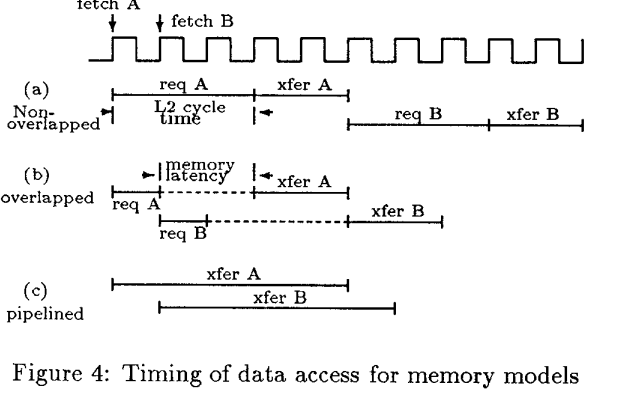
* Three type of architecture

Pure N KB DCache

N KB DCache + 256 entry RPT + 256 entry BPT

N/2 KB DCache + 32N entry RPT +32N entry BPT

**4.2 Memory models**

****

Nonoverlapped: As soon as a request is sent to the next level, no other request can be initiated until the (sole) request in progress is completed. This model is typical of an on-chip cache backed up by a second level cache.

Overlapped: The access time for the memory request can be decomposed into three parts: request issuing cycle, memory latency, and transfer cycles. During the period of memory latency other data requests can be in their request issuing or transferring phases. However, no more than one request issuing or transfer can take place at the same time. This model represents split busses and bank of interleaved memory modules or secondary caches.

Pipelined: A request can be issued at every cycle. This model is representative of processor-cache pairs being linked to memory modules through a pipelined packet-switched interconnection network.

**5 Performance Evaluation**

* Performance assume:

1. No I-cache miss
2. All operations take one cycle
3. No wait on a cache hit
4. The processor stalls on a cache miss until the data is in the cache
5. There is no miss on the next level of the memory hierarchy

**5.1 Metrics**

* We present the results of our experiments by using the contribution to the CPI.

**5.2 Preloading performance for the Nonoverlapped model**

* The “add-cost” organization always performs better than the pure cache scheme(reduce CPI from 10% up to 95%); The “add-cost” organization will always perform better than the “no-cost” .

**5.3 Preloading performance for the Overlapped and pipelined models**

**5.4 Preloading performance VS. Look ahead Limit**

**6 Comparisions with other schemes**

**6.1 Hardware preloading VS. software prefetching**

**6.2 Hardware preloading VS. lock-free cache/decoupled architecture**

**7 Conclusions**

Our scheme solves two aspects of prefetching policy: (1)when to prefetch-at most d(Look Ahead Limit)cycles ahead of current execution, (2)which block to prefetch-the most one likely to be referenced-or not to prefetch at all.

The introduction of a preloading hardware function in a multiprocessor system pose several questions:

1. cache coherence is complicated by the preload requests;
2. the way to capture the regular patterns in a uniprocessor system is not necessarily identical to that of the multiprocessor case, since the loop interations will be spread over several processors
3. memory traffic could increase because of incorrect predictions and more incalidation on preloaded data blocks

# 1994: Evaluating Stream Buffer as a Secondary Cache Replacement

----1994 Subbarao Palacharla

We present two techniques to enhance the effectiveness of Jouppi’s original stream buffers: filtering schemes to reduce their memory bandwidth requirement and scheme that enables stream buffers to prefetch data being accessed in large stride.

**1 introduction**

We consider replacing the secondary cache with Jouppi’s stream buffers.

Stream buffers require much less hardware to implement, yet we find that they can provide performance similar to a large secondary cache for scientific codes.

Memory system efficiency is particularly critical with the context of large-scale parallel machines because the costs of any inefficiencies are magnified by the scale of the system.

**2 Related work**

RPT, keeps currently active load/store instructions and predicts future reference

1191: An effective on-chip preloading scheme to reduce data access penalty, Jean-Loup Baer

1992: Stride directed prefetching in scalar processor, J. W. C. Fu

1992: Prefetch unit for vector operations on scalar computers, Ivan Sklenar

A small prefetching secondary cache backed by high bandwidth Rambus DRAMS

1993: Using Rambus technology in Pentium-based system, Craig Hampel

Evaluated schemes based on the OBL

1982: Cache memories, Alan Jay Smith

1988: Cache operations by MRU change, Kimming So

Stream buffer

1990: Improving Directed-mapped cache performance by the Addition of a Small Full-associative Cache and Prefetch Buffers, Norman P. Jouppi

Instruction cache prefetching in supercomputers

1992: Prefetching in Supercomputer Instruction Cache, J. E. Smith

Compiler prefetching

1991: Software prefetching, David Callahan

1992: Design and Evaluation of a Compiler Algorithm for Prefetching, Todd C. Mowry

**3 Stream buffer**

**4 Methodology**

**4.1 Benchmark and simulation environment**

Trace driven simulation

**4.2 Performance metric**

We use stream hit rate as our primary performance indicator.

**5 Performance of unit stride-only stream**

* Hit rates vary with the number of streams(unified instruction and data buffer):

Majority of the benchmarks show hit rates in the 50%-80% range (values of local hit rates for secondary level caches are in 70%-85%);

Hit rates plateau as the number of streams is increased;

Saturate stream number is related to the number of unique array reference in the program loops (for our benchmarks, seven to eight stream suffice).

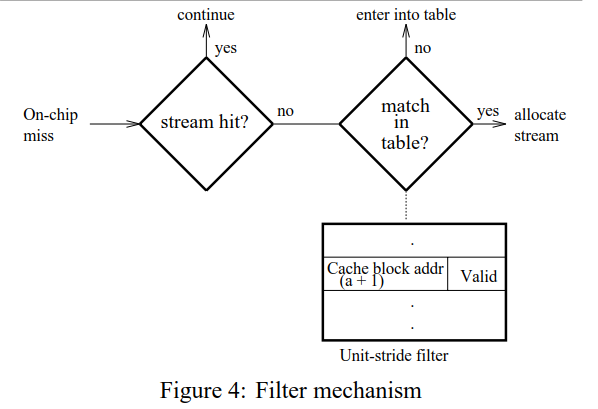
Perform poorly situations: large number of non-unit stride reference; reference data via array indirection

* Compare stream buffer secondary cache

Stream achieve comparable, though perhaps slightly lower, hit rates suggests their use as a viable and cost-effective alternative to huge second level caches.

Compared to secondary caches, streams require more memory bandwidth (EB = stream miss ratio \* depth).

**6 Reducing the Memory Bandwidth Requirements of Stream Buffers**

****

* To reduced wasted bandwidth we have to avoid useless prefetches. One way to avoid unnecessary prefetches is to allocate a stream only when a particular reference shows promise of belonging to a stream.
* The scheme we used:

Filter away isolated references and does not present them to the stream buffers.

* Implemented policy-filter(eight to ten entries is sufficient)

Maintain a list of the N most recent miss address in a history buffer(called filter) but store a+1 for miss address a. For every primary cache miss that also misses in the stream buffers, the miss address is compared with the address stored in the filter.

If there is a hit, this means that there were two references a and a+1 and there is a good possibility that there will be a reference to a+2 and so on. In this case a stream is allocated, and the matched entry in the filter is freed.

If the miss address doesn’t match in the filter, then a+1 is stored in the history buffer(since the filter is not infinite, the new entry might cause an old entry to be replaced).

* Filter advantage and disadvantage

Advantage: It reduces the number of unnecessary prefetchs and it prevents active streams from being disturb. Extra memory bandwidth will be reduced(EB = stream miss ratio \* filter hit ration \* depth)

Disadvantage: the total number of hits could be reduced, since now we allocate a stream only after observing the second reference of a stream accesses.

**6.1 Hit rates for filter-based unit stride streams**

Ten streams(2 depth), a filter of sixteen entries

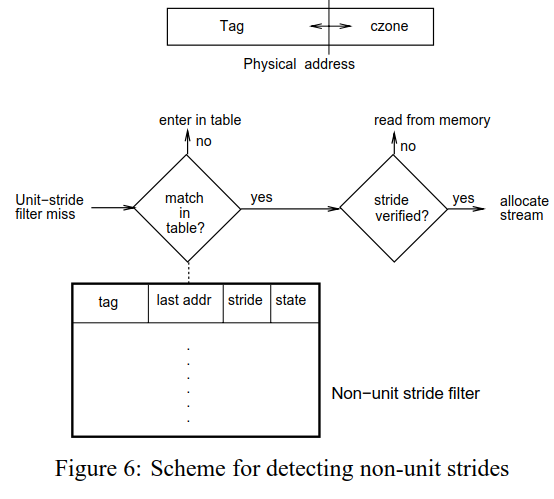
For most of the benchmarks the filter was very effective in reduction EB, often the reduction is more than 50%.

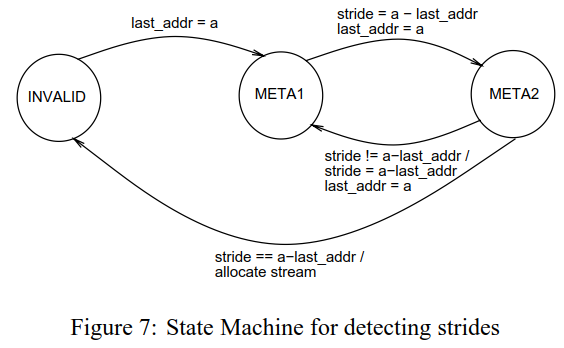
For most benchmarks stream lengths of less than 5 and greater than 20 continue a major fraction of the hits. The programs that have a large concentration of small stream lengths show a greater reduction in hit rate when the filter is used.

From the above results we conclude that a filter may often be a good idea, since in most case it reduces the memory bandwidth requirement of stream for a small or negligible performance hit. At the same time if the program’s memory bandwidth requirement is not high and the memory system is capable of supplying the extra bandwidth, the filter should be deactivated, since the stream buffer hit rate typically falls slightly with the filter.

**7 Detecting non-unit strides**

* The basic idea behind our non-unit stride detection scheme is to dynamically partition the physical address space and detect stride reference within each partition.





* We use the non-unit stride filter behind the unit-stride filter:

Each entry of the non-unit stride filter, in addition to the tag of the partition, has a few state bits, last address and stride fields which are required to implement the stride detecting FSM.

At the end of three consecutive stride references a stream is allocated and entry in the filter id freed.

**7.1 Performance of non-unit stride detecting scheme**

Ten streams, a non-unit stride filter containing sixteen entries

Gains in programs which have a significant number of non-unit stride reference is dramatically.

One has to be careful in selecting the czone size, the optimal size for the czone is (a little more than) twice the stride of the reference. Since the size of the czone depends on the stride and array dimensions, it is possible for the programmer or the compiler to set it to a suitable value.

**8 Comparison with second level caches**

Stream buffers typically scale better than secondary caches.

**9 Conclusions**

For the majority of the benchmarks we studied, a hit rate of greater than 60% using only 30% extra main memory bandwidth is achieved using ten streams.

# 1997: Stride-directed Prefetching for secondary cache

----1997 Sunil Kim

**1 introduction**

* 只给 L2-cache增加预取策略，理由：

off-chip的L2-Cache可以加入较大规模的预取器件；

一级cache的预取复杂度会可能会增加处理器周期时间；

二级cache相比于一级cache更能容忍cache污染；

* 91,92, 93年的的存储器访问流是通过指令地址来检测的，但这种基于指令地址的预取比较适用于L1 cache，不适用于L2 cache，理由：

L2 cache只能看到L1 的miss，看不到所有的数据访问，this make it difficult to detect a constant stride due to locality or due to cache conflicts in L1 cache;

对于片外的L2 cache，获取指令地址信息成本比较高。

* （94，95年文献中提出的L2的流检测方式）L2 method to detect reference stream:

Separate concentration zone, reference falling into the same zone are consider to be in the same stream and are used to calculate the stride.(zone size is delegate to either a compiler or a programmer)；

Minimal delta：detect a stream by finding two memory reference with the smallest address distance among last few memory references.

* 本文提出一种新的L2 cache的流检测方式：loop-based prefetching。它不需要指令地址，且硬件开销适中。
* 对以下四种cache组织的性能进行比较：

Add a small cache to prefetch hardware

Traditional cache

Stream buffer prefetching without a cache

Prefetch with a small cache

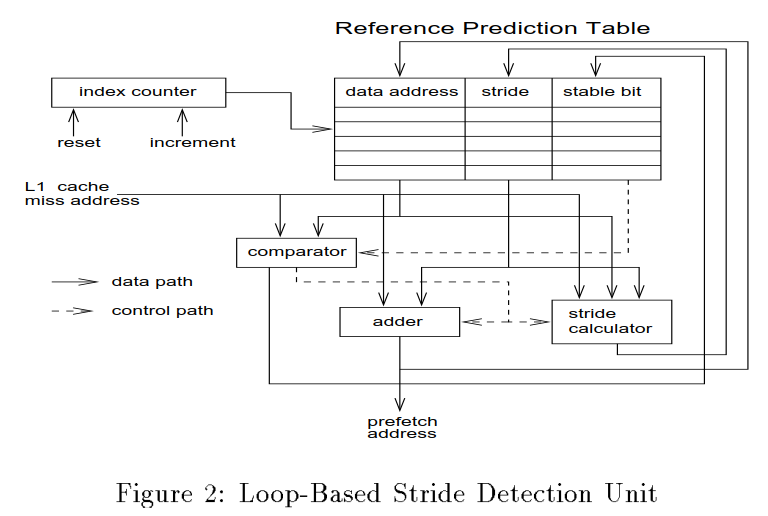
**2 Loop-based Prefetching and its implementation**

循环体中的访存指令会被重复执行，新一轮循环中的访存指令的执行顺序往往和上一轮循环的执行顺序保持一致。如果能获取每一轮循环的开始标识（比如特定的访存地址，比如分支的回跳），就能通过访存指令距离循环开始标识的距离来区分存储访问流，这个相对距离作为RPT的访问指针，每一轮新的循环都会将指针counter置0。如果指针所指项有预取请求，则会在L1请求前发起预取操作。如果计数器溢出，即循环体访存指令数目超出RPT项数，所有后续访问会被忽略。

因为L2的循环预取硬件看不到在L1中命中的操作，因此L2 prefetching unit uses only L1 misses to increment the index to RPT and keep trying to acquire a constant stride form L1 miss mapped to the same RPT entry。和其他预取机制一样，会因为L1 cache的屏蔽而使其性能变差。

这种方法可以拓展到多重嵌套循环，但是需要提供更多的分支执行信息。在本文我们只关注最内层的程序循环，因为绝大多数的访存都在内层循环中。

实现方式：



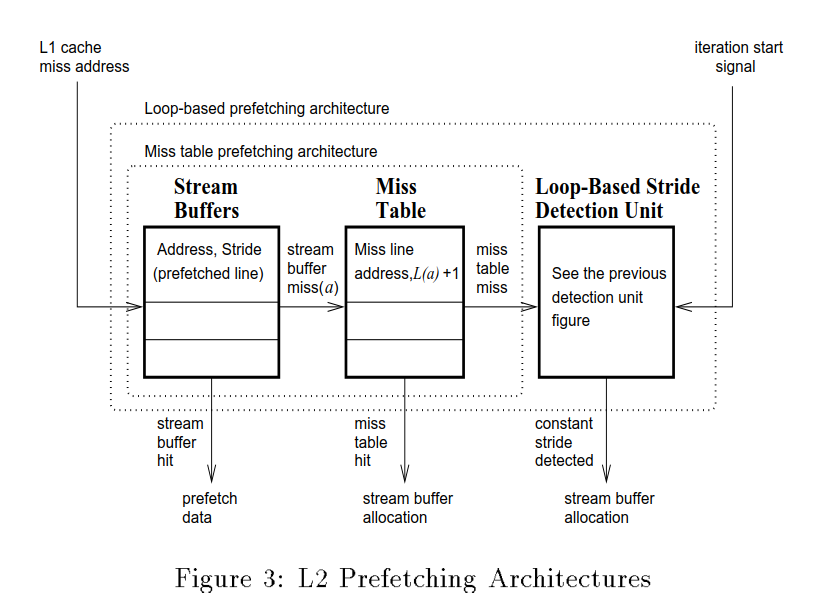
**3 performance Evaluation Methodology**

* Trace-driven simulation，一个理想的ICache，只有load会被预取，循环开始的标识是通过插入特殊指令。
* 使用PPMR(Prefetch prediction miss ratio)和memory traffic 来评估预取策略的性能。
* 用于对比的三种预取结构：

stream buffer + MTP(Miss Table Prefetching)

stream buffer + MTP(Miss Table Prefetching) + LBP(Loop-based prefetching)

stream buffer + MTP(Miss Table Prefetching) + IAP(Instruction-address-based Prefetching)



**4 Experimental Results and Analysis**

* IAP 应用于L1和L2的PPMR的对比：stride-directed prefetching即使能够做到精确的流检测，其在L2的预测准确率也要比L1差很多，因为L1对L2有存储访问的屏蔽作用。
* Stream buffer number：16，64，256。Even the smallest size used here is larger than what has been previously studied since we believe that the extra locality can be exploited with more buffers.

Miss table size：32. Increasing the size beyond 32 did not improve the prediction miss rate significantly but hurt the performance in some case.

RPT entries: 64，in fact no more than 33 entries were used in any of our benchmark.

* 三种预取结构的stream buffer miss ratio对比：

随着stream buffer数量的增加，miss ratio曾呈现低趋势（16-64-256）；

LBP展现出比MTP相当或者更少的miss rate, by detecting long stride access;

IAP while theoretically capable of detecting memory streams more accurately than the loop-based stride detection unit performs worse than MTP and LBP(LBP 探索了更多的cross-stream locality).

* Cross-stream locality：stream buffer也可能会命中不是正确分配的流，或者命中其他流。

MTP和LBP可能会有这样的情况发生，但IAP因为流的精准定位，反而失去了这部分的收益，所以表现出的性能反而不如MTP和LBP。（误打误撞也能提高性能）

* Stride detection is more difficult at L2 and not as effective as the L1 because L1 misses which arrive at L2 may not have a constant stride. Therefore, prefetching methods which can take advantage of good cross-stream spatial locality should be used for L2 prefetching.
* The addition of even a small L2 cache improves the performance for all benchmarks (+prefetching).
* The LBP system can attain up to 40% higher hit rates in half of the benchmarks than a 16-times larger non-prefetching L2 cache.
* Traffic increase under prefetching: in most case the increase is less than 10%, the low increase due to the fact that prefetches are not start until a stable stride is detected and only one cache line is prefetched each time.

**5 conclusions**

实验表明，使用cross-stream locality能有效提高L2 cache 预取的效力；LBP+MTP能更好地提高预取性能；增加stream buffer的数量可以提高性能；附加一个小的L2 cache能进一步提升LBP预取的性

**参考文献**

1990：An Effective On-chip Preloading scheme to reduce data access penalty, Jean-Loup

1991：Stride Directed Prefetching in scalar processor, Edward H. Gornish

1993：Speculative Prefetching, Y. Jegou

1994：Evaluating stream buffers as a secondary cache replacement, Subbrao Palacharla

1995：How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processor? Keith Faekas

# 1997: Prefetching Using Markov Predictors

----1997, Doug Joseph

**1 Introduction**

* A memory prefetching mechanism attempts to provide data before the processor requests the data.
* Three metrics used to compare memory prefetchers

Coverage: the fraction of memory request that were supplied by the prefetcher rather than being demanded-fetched.

Accuracy: fraction of the prefetched cache lines offered to the processor that were actually used.

Timeliness: indicates if the data offered by the prefetcher arrives before it is needed, but not so early that the data must be discarded before it can be used.

* Prefetching mechanisms design steps

First, the architect envisions a “model” describing the way that the programs behave when accessing memory, and then attempts to construct a physical realization of that model that provides suitable prefetch coverage.

* In memory-level simulations, we find that this prefetching mechanism can reduce the memory overhead to the CPI by 54%.

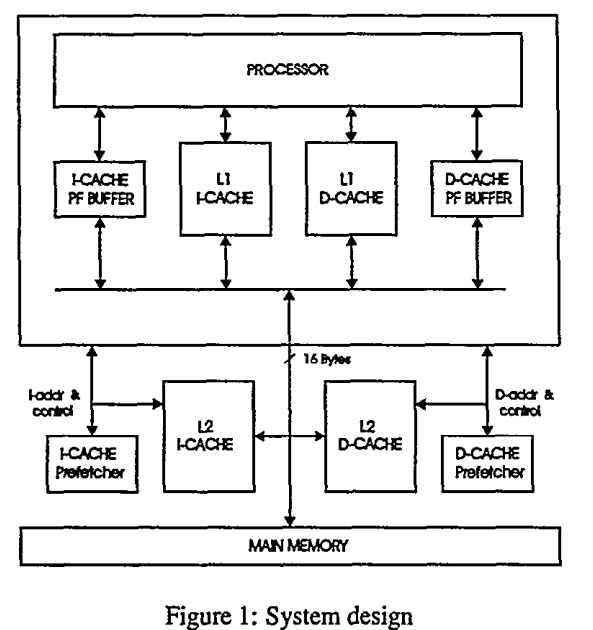
1. **Prefetcher Design and Implementation**

* Hardware prefetching is a prediction process: given some current prediction state, the prefetcher guesses what a future memory reference may be and requests that location from the memory subsystem.
* Prediction information source

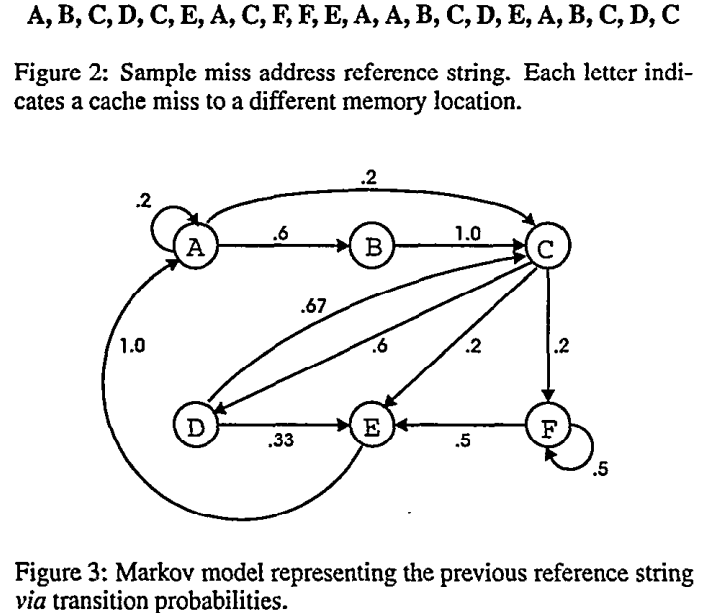
Address access stream

Miss address stream

* Schematic design for the prefetcher



* 1. **Modeling Memory References via Markov Processes**



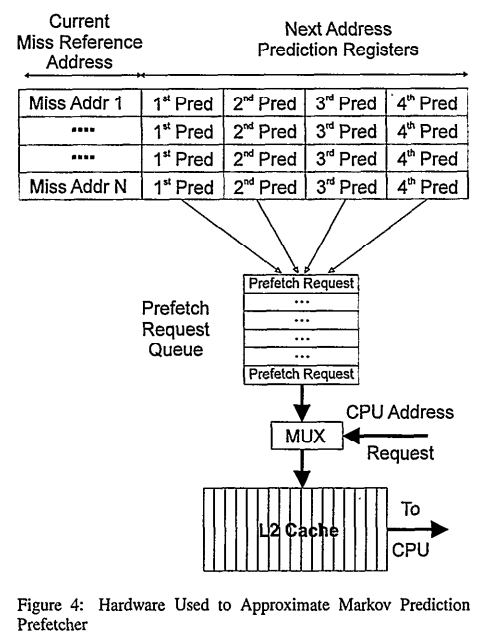
* 1. **Realizing the Markov Prefetcher in hardware**
* Problems encounter in “pure” Markov model

(1)In practice programs don’t repeat exactly the same reference patterns from one execution to another, and the transition probabilities “learned” in one execution may not benefit another.

(2)It is difficult to efficiently represent a pure Markov model in hardware because each node may have an arbitrary degree and the transition probabilities are represented as real values.

(3)Programs reference millions of addresses and it may not be possible to record all references in a single table.

* Hardware implementation



When the current miss address matched the index address in the prefetch table, all of the next address prediction registers are eligible to issue a prefetch. Each prefetch request has an associated priority. Prefetch addresses are stored in the prefetch request queue, and higher priority requests can dislodge low-priority request. The prefetch request queue contends with the processor for the L2 cache, and the demand fetches from the processor have higher priority. Thus, after a series of prefetches, the prefetch request queue may be full, and lower-priority requests will be discarded.

Once a fetch request is satisfied by the L2 cache, it is placed in the on-chip prefetch buffers. We model the on-chip prefetch buffers as a 16 entry full associative FIFO buffer. When the processor queries it, all entries are associatively searched in one cycle. If a match is found, it is relocated to the head of the FIFO, and all the entries from the head to the vacated slot are shifted down by one. The FIFO is also searched when updated to avoid duplicate entries. If there are no duplicates when adding an entry, an empty slot is filled, or if there are no empty slots the last slot is replaces.

1. **Prior Work**

Static predictor: almost all static predictors rely on the compiler to determine possible L1 cache misses and embed the information into the code in the form of prefetch instructions.

1992: Design and Evaluation of a Compiler Algorithm for Prefetching, T. C. Mowry

1995: Software Prefetching in Pointer and Call Intensive Enviornments, M. H. Lipasti

1995: Cache Miss Heuristics and Preloading Techniques for General-purpose Programs, T. Ozawa

Stride prefetchers

1992: Reducing Memory Latency via Non-blocking and Prefetching Caches, T. F. Chen

Stream buffers

1990: Improving Directed-mapped Cache Performance by the Additional of a Small Fully Associative Cache and Prefetch Buffers, N. Jouppi

1994: Evaluating Stream Buffers as a Secondary Cache Replacement, S. Palacharla

1994: Complexity/performance tradeoffs with non-blocking loads, K. I. Farkas

Indirect Stream Detector

1995: Examination of a Memory Access Classification Scheme for Pointer-intensive and Numeric Programs, S. Mehrota

Correlation-Based Prefetching

1976: Dynamic Improvements of Locality in Virtual Memory Systems, J. L. Baer

1989: Prefetching System for Cache Having a Secondary Directory for Sequentially Accessed Blocks, J. Pomerene

1995: Generalized Correlation Based Hardware Prefetching Caches, M. J. Charney

1996: Distributed Predictive Cache Design for High Performance Memory System, T. Alexander

1. **Experimental Design and Simulation Study**

* Metric

In this study, we use a single metric, the fraction of first-level cache misses, to characterize both coverage and accuracy.

We simulating a non-speculative processor with a non-speculative processor with a detailed memory model and comparing the MCPI (memory cycle-per-instruction), This represents the average number of CPU stalls attributed to the memory subsystem.

* Baseline configurations

All non-memory instructions execute in one cycle

A single-cycle on-chip 8KB L1 data cache and an 8KB L1 instruction cache, each cache has 8-entry single-cycle victim buffers and 32-byte lines. The L1 data cache also has a single-cycle 8-entry write buffer and uses a write-around policy.

Te second-level cache were multi-cycle, 4-bank, direct mapped, lockup-free 4MB I and D caches, with 128 byte lines. The L2 data cache uses a write-back with write allocate policy and had one 8-entry address request queue per bank. The address and data buses have a latency of four cycles. Each cache bank has a separate address bus to each L2 cache bank but just one L2 data bus shared by all banks. The L1-L2 bus bandwidth is 8 byte/cycle, the access latency of a bank is 24 cycles and L2-L3 bus bandwidth is 4 bytes/cycle.

* Stream buffer configurations

When simulating stream buffers, we used eight three-entry stream buffers, with associative lookup, non-overlapping stream allocation, and allocation filters. Each stream buffer has single cycle access.

When simulating stride prefetchers, we used a stride prefetcher with 16 entry fully associative stride detection table. Access is also single cycle.

* Correlation and Markov prefetchers configurations

We configured the Markov and Correlation prefetchers to use less memory than the corresponding demand-fetch cache or the stride and stream prefetchers. For the Markov and Correlation prefetchers, we used a 1MB data prefetch table and a 2MB data cache. For the instruction cache, we also used a 2MB instruction cache and 1MB prefetch table for the prefetch configurations.

* 1. **Benchmark Applications**

Many important technical and commercial applications give rise to unstructured workloads. Technical applications involving large sparse arrays of data often store such data in a compressed format and access that data via indirection arrays. Another common source of unstructured access patterns in technical and commercial workloads is the use of pointer connected structures.

Eight test programs: four of the traces used were generated from unstructured technical codes, and four were from commercially oriented workloads.

1. **Performance Comparison**

We compare the performance of the Markov prefetcher using four prefetch addresses and an LRU prioritization policy against other prefetch mechanisms.

* Fanout choose for Markov prefetcher

In most applications that we examined, four prefetch predictors provide a reasonable balance between coverage and accuracy for the data cache. Only two preftchers were needed for the instruction cache, because there are fewer successors for a given instruction reference.

* Transaction probabilities choose for Markov prefetcher

We considered two methods to prioritize the references. In the first, we use the true transition probability and in the second, we used a simple LRU algorithm. The LRU algorithm out-performed the true transition probability in every application, and is much easier to implement. The LRU prioritization method not only requests the most frequently needed items, but that it also requests those that are needed soonest.

* Six compared prefetching

(1)Stream prefetching

(2)Stride prefetching

(3)Correlation prefetching

(4)Markov prefetching

(5)Stride, stream and Markov in parallel

(6)Stride, stream and Markov in series

* Test data show

Stream prefetchers provide better coverage than stride prefetchers, but do so by using considerably more bandwidth. The correlation prefetcher provides still better coverage and the Markov prefetcher provides the best coverage over the set of applications. However, as with stream buffers, increase in coverage comes at the price of increased memory bandwidth demands, often five-fold than that of the stream buffers.

* 1. **Limiting the Prefetch Bandwidth**

# 2004: Data Cache Prefetching Using a Global History Buffer

----2004, Kyle J. Nesbit

**1 Introduction**

* The simplest prefetch methods are sequential

1978: Sequential Program Prefetching in Memory Hierarchies, A. J. Smith

1990: Improving Direct-mapped Cache Performance by Addition of a small Full Associative Cache and Prefetch Buffers, N. Jouppi

1994: Evaluating Stream Buffers as a Secondary Cache Replacement, Subbarao Palacharla

* More advanced prefetch methods use tables to record history information related to data access

1992: Stride Directed Prefetching in Scalar Processors, J. W. C. Fu

1994: Effective Hardware Based Data Prefetching for High-performance Processor, T. Chen

1994: Evaluating Stream Buffers as a Secondary Cache Replacement, Subbarao Palacharla

1997: Stride-directed Prefetching for Secondary Caches, S. Kim

1999: Prefetching Using Markov Predictors, D. Joseph

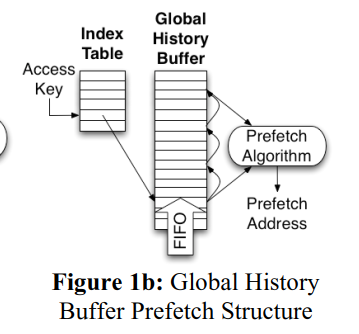
2001: Dead-block Prediction and Dead-block Correlating Prefetchers, A. C. Lai

2002: Gong the Distance for TLB Prefetching: An application-driven Study, G. B. Kandiraju

2002: Using a User-level Mmeory Thread for Correlation Prefetching, Y. Solihin

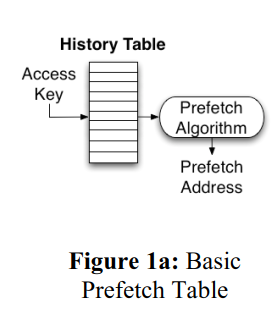
2003: TCP Tag Correlating Prefetchers, Z. Hu

* We proposed structure: all address history is held in a FIFO table, the GHB(Global History Buffer), with al global miss addresses being placed in the table at the bottom and removed from the top. GHB history information is maintained in linked lists, which are access indirectly via a hash table.



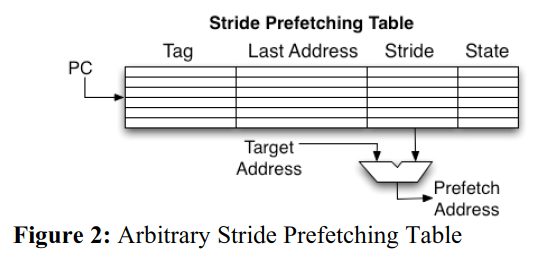
* We focus on the lowest level data cache because modern out-of-order processors can tolerate most L1 data cache misses with relatively little performance degradation.

**2 Table-based Prefetching**



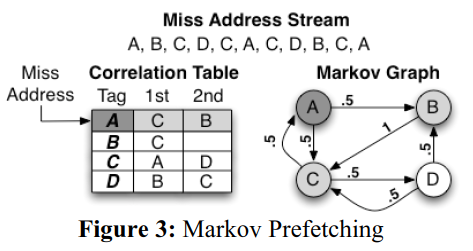
**2.1 Stride Prefetching**

The PC of a load instruction indexes the table. Each table entry holds the load’s most recent stride, last address, and state information describing the stability of the load’s recent stride behavior. When a prefetch is triggered, address a+s, a+2s, …, a+ds are prefetched, where a is the load’s current target address, s is the detected stride and d is the prefetch degree, an implementation dependent prefetch look-ahead distance; more aggressive prefetch implementations will use a higher value for d.



**2.2 Markov Prefetching**

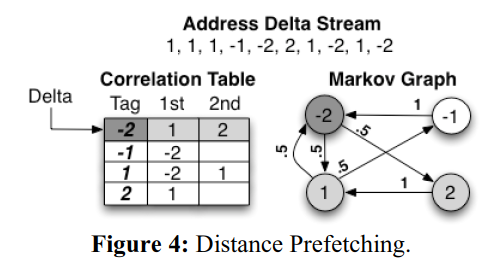
Markov prefetching is an example of a correlation prefetching method. Correlation prefetching uses a history table to record consecutive address pairs. When a cache miss occurs, the miss address indexes the correlation table, each entry in the Markov correlation table holds a list of addresses that have immediately followed the current miss address in the past. When a table entry is accessed, the member of its address list are prefetched, with the most recent miss address first.



**2.3 Distance Prefetching**

Distance prefetching uses the distance between two consecutive global miss addresses, an address delta, to index the correlation table. Each correlation table entry holds a list of deltas that have followed the entry’s delta in the past.

Distance prefetching is considered a generalization of Markov Prefetching because one delta stream correlation can represent many miss address correlations. By generalizing Markov Prefetching, distance prefetching is capable of prefetching most of the reference patterns that Markov Prefetching can. Plus, with the data it has available, it can also detect and prefetch delta access patterns that occur in the global miss address stream.



**3. Global History Buffer Prefetching**

* Prefetch table’s inefficiency

1. Table data can become stale, and consequently reduce prefetch accuracy
2. Table suffer from conflicts that occur when multiple access keys map to the same table entry
3. Table have a fixed amount of history per entry

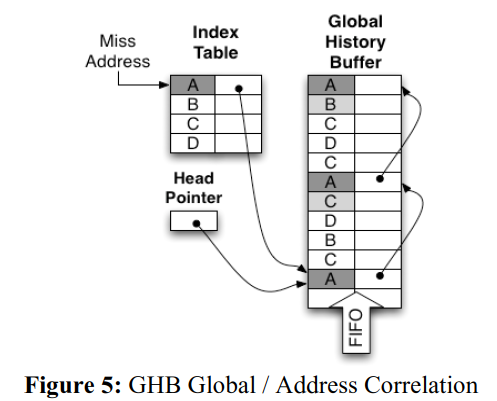
* The GHB prefetching structure has two level

1. IT(Index Table): accessed with a key as in conventional prefetch tables. The key may be a load instruction’s PC, a cache miss address, or some combination. The entries in the IT contain pointers into GHB.
2. GHB(Global History Table): an n-entry FIFO table that holds the n most recent L2 cache miss address. Each GHB entry stores a global miss address and a link pointer. The link pointers are used to chain the GHB entries into address lists. Each address list is the time-ordered sequence of addresses that have the same IT key.

* Prefetching methods taxonomy: X/Y, where X is the key used for localizing the miss address stream and Y is the mechanism used for detecting addressing patterns.

1. Localizing methods: PC(program Counter) and G(Global)
2. Detection mechanisms: CS(Constant Stride), DC(Delta Correlation), AC(Address Correlation)

**3.1 Example: Markov Prefetching**



G/AC: when an L2 cache miss occurs, the miss address indexes the IT. IF there is a hit in the IT, the IT entry will point to the most recent occurrence of the same address in the GHB. This GHB entry is also at the head of the linked list of other entries with the same miss address. For each entry in this linked list, the next entry in the FIFO ordered GHB is the miss address that immediately followed the current miss address when it occurred in the past. There “next” global miss address are prefetch candidates.

**3.2 Example: Stride Prefetching**

PC/CS: Using the PC of a load instruction as the index into the IT, the address list created is the sequence of addresses for the given PC. The load’s stride can be calculated by computing the differences between consecutive entries in the address list. If a constant stride is detected, then addresses a+s, a+2s, …, a+ds are prefetched—where a is the current miss address, s is the detected stride and d is the prefetch degree.

**3.3 Implementation**

* Access fashion: For each new miss, the GHB is updated in FIFO fashion. The miss address is placed into the GHB entry pointed to by the head pointer, and its link entry is given the current value in the IT. The IT link entry is then updated with the head pointer, which points to the newly added entry. Finally, the head pointer is incremented to point to the next GHB entry.
* Advantage

1. GHB FIFO naturally gives table space priority to the most recent global history; the GHB naturally allocates more chip-are to events that have occurred more recently and more often
2. The Index table and GHB are sized separately
3. The ordered global history can be used to create more sophisticated prefetching methods

* Disadvantage

Collecting prefetch information requires multiple table accesses

**3.4 Generalized Correlation Prefetching**

The Markov model can serve as a basis for a number of global correlation prefetching methods: the nodes are addresses and acr connecting two nodes indicates the probability that one address is followed by the other global miss address stream.

Width prefetching: start at a node and prefetch using one or more adjacent nodes – but only the immediately adjacent nodes.

Depth prefetching: beginning with the current miss address, the sequence of the most likely arcs is followed, with prefetching initiated at each node along the path.

**3.5 Local Delta Correlation**

PC/DC, uses delta pairs as the correlation key.

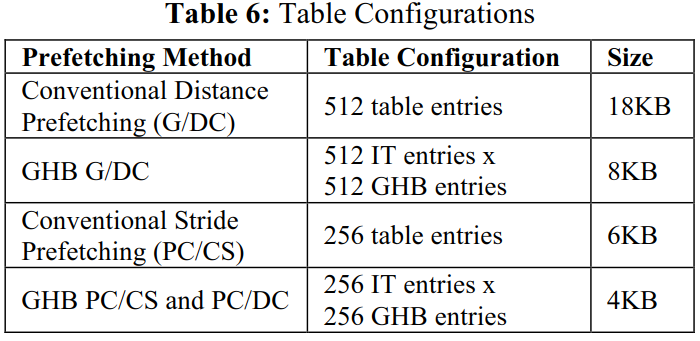
**4 Performance Evaluations**

**4.1 Simulation Methodology**

* A subset of the SPEC CPU2000 benchmark suite
* SimpleScalar was used for collecting performance data
* To eliminate the need for additional prefetch structures, prefetched lines are placed directly into the L2 cache. To keep prefetched lines from modifying the “natural” L2 cache miss address stream, one bit prefetch tags are added to the L2 cache lines. When a prefetched line is written into the L2 cache, its prefetch tag is set; when a cache access hits a prefetched line with a set prefetch tag, the prefetch tag is cleared, and the access’s memory address is sent to update the prefetch structures as if it were an L2 cache miss.

**4.2 Table Configurations**

* To compare the performance of different sized tables we use the IPC improvement harmonic mean of the benchmarks. Optimal GHB G/DC methods table configuration and optimal GHB PC local methods table configuration:



**4.3 GHB Prefetch Performance**

* GHB prefetch performance: for a prefetch degree of four, GHB G/DC hybrid prefetching outperforms conventional Distance Prefetching by 13%.
* PC local prefetching: for a prefetch degree of four, GHB PC/DC consistently performs better than constant stride preftehcing. At a prefetch degree of sixteen, conventional stride prefetching performs 1% better and consumes 1% more traffic than GHB PC/CS(equivalent performance).

**5 Conclusions**

The first GHB prefetching methods, GHB G/DC hybrid, is shown to have a 20% IPC improvement over its conventional counterpart, Distance Prefetching, and it can reduce memory traffic by 90%. The second GHB method, GHB PC/DC, is shown to have 6% IPC improvement over its conventional counterpart Stride Prefetching.

# 2004: AC/DC: An Adaptive Data Cache Prefetcher

----2004 Kyle J. Nesbit

C/DC(CZone prefetching with Delta Correlations) yields an average performance improvement of 23% when compare with no prefetching. AC/DC adaptive reconfiguration provides additional performance improvements of 4% over C/DC.

**1 Introduction**

As the processor-memory latency gap continues to increase, there is a continuing need for development and refinement of prefetch methods. In this paper, we propose an innovative method for cache prefetching aimed specifically at prefetching from main memory.

We use and adaptive tuning algorithm that operates along two dimensions- for each program phase it dynamically adjusts both the CZone size and prefetch degree to find near-optimal configuration.

**2 Related Work**

**2.1 Stride Prefetching**

Stride prefetching detects sequences of addresses that differ by a constant value, and launch prefetch requests that continue the stride pattern.

1990: Improving directed-mapped cache performance by addition of a small fully associative cache and prefetch buffers, N. Jouppi

1992: Stride directed prefetching in scalar processors, J. W. C. Fu

1994: Evaluating Stream Buffers as a Secondary Cache Replacement, S. Palacharla

1995: Effective hardware based data prefetching for high-performance processors, T. Chen

1997: Stride –directed Prefetching for secondary caches, S. Kim

2001: POWER4 System Microarchitecture, J. M. Tendler

**2.2 Correlation Prefetching**

Correlation Prefetching methods look for address sequence patterns in order to predict future cache behavior.

1999: Prefetching Using Markov Predictors, D. Joseph (global miss address prefetching)

2002: Going the Distance for TLB Prefetching, G. Kandiraju (distance prefteching)

2003: TCP Tag Correlating Prefetchers, Z. Hu (two-level correlation prefetching)

2004: Prefetching with a Global History Buffer, K. Nesbit (adaptation correlates deltas correlation prefetching)

**2.3 Prefetching with a Global History Buffer**

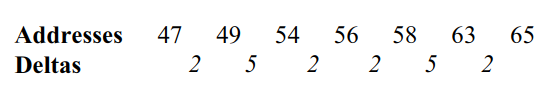
The FHB is an n-entry FIFO table(implemented as a circular buffer) that holds the n most recent L2 miss addresses. Each GHB entry stores a global miss address and a link pointer. The link pointers chain the GHB entries into time ordered address lists. An IT(Index Table) holds the initial pointers to the linked lists. The IT is accessed via some key; depending on the key that is used, a number of history-based prefetch methods can be implemented.

2004: Prefetching with a Global History Buffer, K. Nesbit

2004: MicroLib: a case for the quantitative comparison of micro-architecture mechanisms, Daniel Gracia Perez

**3 CZone Delta Correlation Prefetching**

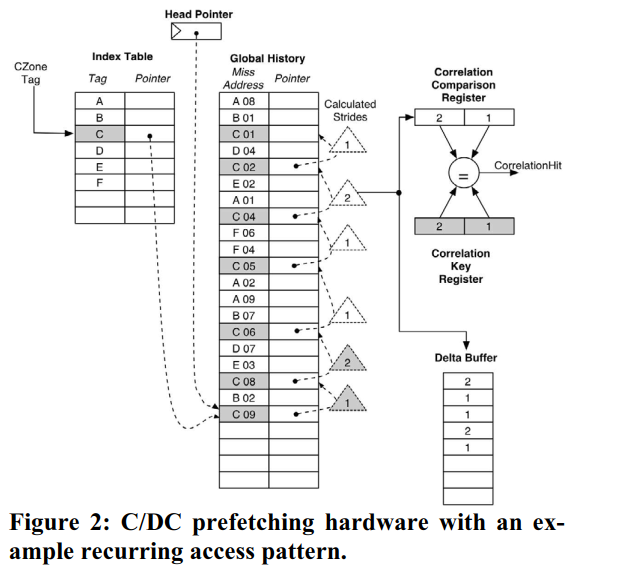
CZone prefetching divides memory into fixed size zone, typically powers of two. Then, it looks for patterns within each zone.

(5,2)—(2, 5)—67prefetch—72prefetch

In general, the sequences of deltas used for correlation can be any length. We undertook a preliminary study that indicated that pairs of deltas are a good choice.

**3.1 Implementation Details**

To the basic GHB structure, we add a small delta buffer, a correlation key register, and a correlation comparison register.



When a load miss the L2 cache, the CZone tag of the miss address is used to access the IT. If the CZone tag hits in the IT, the IT entry points to the list of preceding miss addresses that are in the same CZone.

The correlation mechanism then compares pairs of deltas in the miss address stream. The first delta in the address sequence is computed and shifted into the correlation key register. On the next cycle, the next element of the linked list is accessed, and the second delta is computed and shifted into the correlation comparison register. After the second delta, the linked list is walked and deltas are shifted into the correlation comparison register. At each step the comparison register and the key register are compared. If the registers match, a correlation has been detected and prefetching is triggered.

As deltas are computed and shifted into the correlation comparison register, they are also shifted into the delta buffer and are held. Prefetch addresses are generated by accessing the delta buffer tail and processing toward the head until the desired prefetch degree has been reached. To compute the sequence of prefetch addresses the delta values are consecutively added to the miss address. The first two deltas in the delta buffer are ignored, because they are part of the current correlation.

In addition to delta pair detection, the C/DC prefetcher also has a mechanism to quickly identify constant stride memory access.

**3.2 Discussion**

Recent prefetch research were designed to prefetch into the lowest level cache, because modern out-of-order processors can tolerant most L1 data cache misses with relatively little performance degradation.

Three main advantages to the C/DC prefetch:

1. Localizing address streams based on CZones is well-suited for L2 prefetching
2. The use of a FIFO GHB naturally gives priority to recent program behavior and eliminates “stale” history by evicting the oldest miss address history. And with CZone prefetching, each GHB entry stores only the low-order bits of the miss address, because the high order bits are implied by the CZone tag that accesses the IT.
3. The use of delta pair correlation is more general than using constant strides.

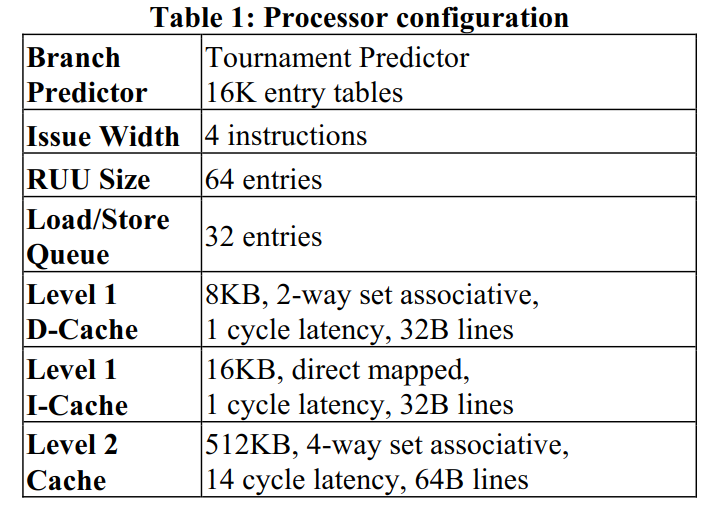
Disadvantage of the GHB: the time required to walk the linked list and perform correlations. This delay id mitigated by the long latencies of cache misses to main memory-on the order of hundreds of cycles.

**4 Evaluation of C/DC Prefetching**

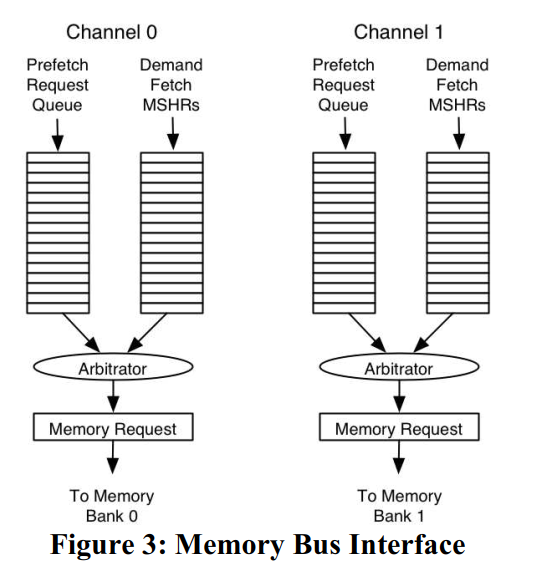
**4.1 Simulation Method**

Baseline

A modified version of Simple Scalar 3.0 was used for modeling an out-of-order processor.

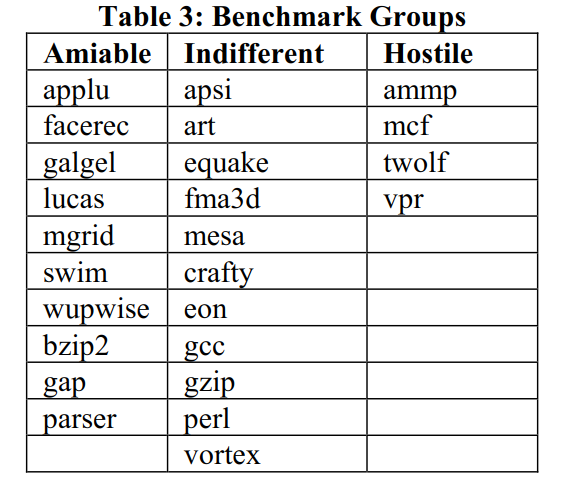


We add a DDR memory model to SimpleScalar. The memory bus has two memory channels, which is representative of modern high-end desktop system. Each memory channel has a 16-entry miss status handling register queue for demand fetches and a 16-entry request queue.



Benchmark

SPEC CPU2000



Measure

IPC, Memory Utilization

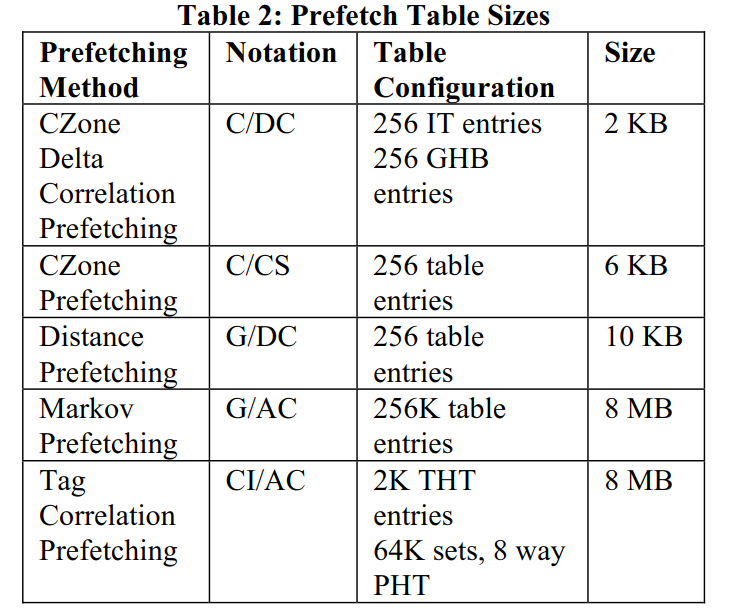
Compare

CZone Prefetching (1994, S. Palacharla)

Distance Prefetching (2002, G. Kandiraju)

Tag Correlation Prefetching (2003, Z. Hu)

Markov Prefetching (1999, D. Joseph)



To keep prefetched lines from modifying the “natural” L2 demand miss address stream, one bit prefetch tags are added to the L2 cache lines. When a prefetched line is written into the L2 cache, its prefetch tag is set. When a cache access hits a prefetched line with a set prefetch tag, the prefetch tag is cleared, and the access memory address is sent to update the prefetch structures as if it were an L2 cache miss.

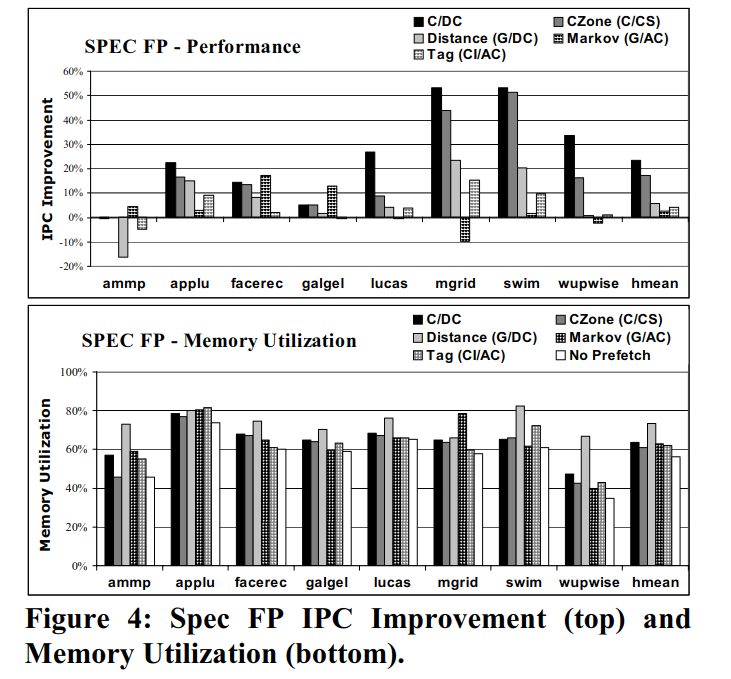
The baseline configuration for each method (except Tag Correlation Prefetching) has a prefetch degree of four. For the baseline prefetch degree of four, we simulated the SPEC CPU2000 benchmark suite with different CZone size and chose a baseline CZone size (64KB) that was near optimal.

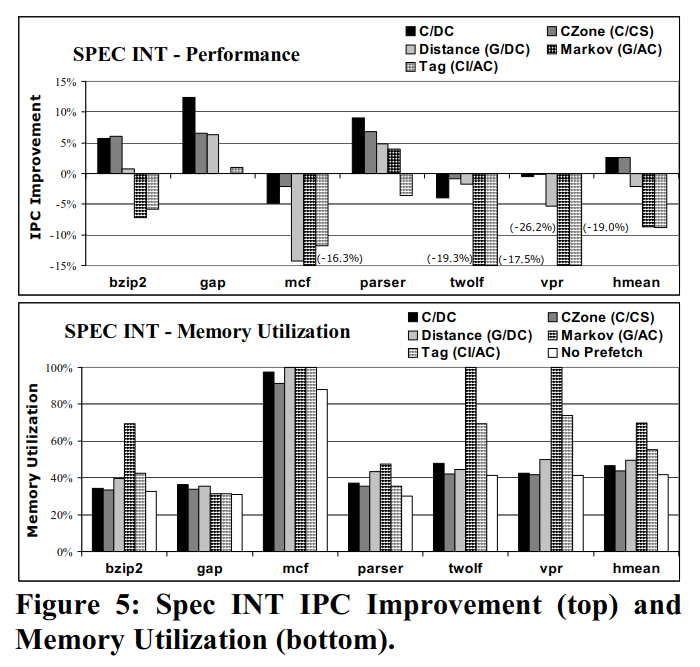
**4.2 Results**

The benchmarks are divided into three groups.

1. Amiable benchmarks are those where at least one prefetching method studied improves performance by more than 5%, all amiable benchmarks have less than 75% memory utilization without prefetching.
2. Indifferent benchmarks are hose whre none of the prefetching methods hurt performance, but no method improves performance by more than 5%.
3. Hostile benchmarks are those where prefetching tends to degrade performance; this typically occurs when a benchmark has phases with very high memory utilization.

IPC improvement and memory utilization for the amiable and hostile SPEC2000 benchmarks:





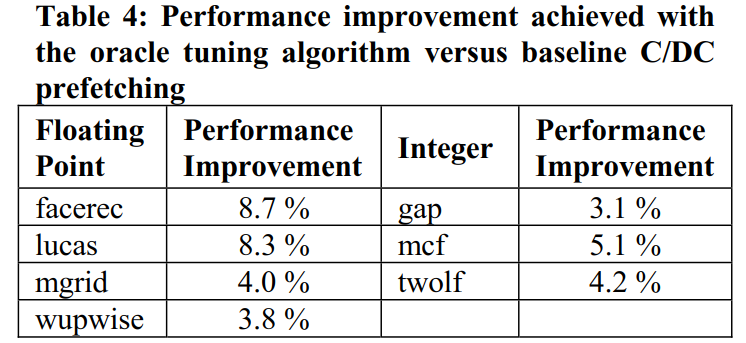
For workloads that are sensitive to memory contention inaccurate prefetches result in higher memory utilization that degrades performance of all the studied prefetchers.

**5. Adaptive Prefetching**

Different programs use different data structures and access patterns. Conseuquently, the optimal CZone size and prefetch degree vary across programs. Even within a program, the data access patterns might change as the program goes through various phases of execution.

**5.1 An Oracle Tuning Algorithm**

In order to evaluate the performance potential of adaptively tuning the CZone size and prefetch degree, we first implemented an oracle tuning algorithm.



The oracle tuning algorithm is highly effective and can provide additional performance benefits of 2% to 9% beyond the baseline C/DC prefetch methods.

* 1. **A phased-based Tuning algorithm**

Tuning algorithms have been proposed to control dynamically configurable hardware structures such as:

**Cache**:

2000, Memory hierarchy reconfiguration for energy and performance in general purpose architectures,

R. Balasubramonian

**Branch predictors:**

2001, Performance for selected SPEC CPU2000 benchmarks, J. F. Cantin

**Pipelines:**

2001, Power and energy reduction via pipeline balancing, R. Bahar

**Program phase detection:**

2002, Dynamic Microarchitecture adaptation via co-designed virtual machines, J. E. Smith

2002, Managing Multi-Configuration Hardware via Dynamic Working Set Analysis, J. E. Smith

2003, Phase Tracking and Prediction, T. Sherwood

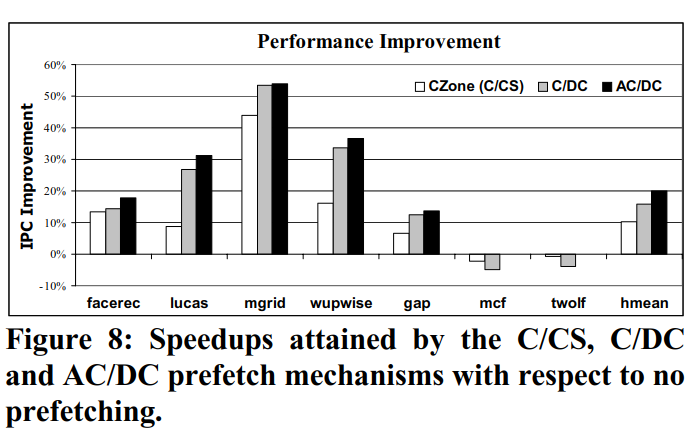
2003, Positional Adaptation of Processors: Application to Energy Reduction, M. Huang

* 1. **Optimizations**

Because program phases have different lengths and periodicities, a single tuning interval is not necessarily suitable for all programs. Consequently, when tuning for a given program, it is best to first adapt the length of the tuning interval to the program.

The performance variation with CZone size is roughly the same for any given prefetch distance, and vice versa. Thus, the tuning of CZone size and prefetch degree can be done independently, e.g. first optimize CZone, then optimize prefetch degree-leading to a significant reduction in tuning combinations.

* 1. **Evaluation**



We evaluated the tuning algorithms by simulating each benchmark for 4 billion instructions. The simulator is fast forward by 1.75 billion instructions, after which the tuning algorithm searches for the best tuning interval for 250 million instructions. Performance is then measured over the next 2 billion instructions.

C/DC outperforms C/CS on most benchmarks, except the hostile ones. On hostile benchmarks, C/CS leads to smaller performance losses than C/DC mainly due to fewer prefetches generated. AC/DC, on the other hand, outperforms C/CS consistently on all benchmarks. On hostile benchmarks, AC/DC often turns off prefetching, thereby eliminating performance loss. On average, AC/DC improves performance by 10% over C/CS.

AC/DC also improves performance over C/DC on all benchmarks. In the case of amiable benchmarks, the performance improvements are a result of adapting the prefetch parameters. In hostile benchmarks, the improvements are a result of turning off prefetching. On average, AC/DC provides 4% performance improvement over C/DC.

**6 Conclusions**

Adaptivity provides performance improvements no only by tuning CZone size and prefetch degree, but it also provides benefits by tuning off prefetching in situations where performance is degraded.

# 2005: On the importance of optimizing the configuration of stream prefetchers

----2005 Iiya Ganuov

# 2009: Spatio-Temporal Memory Streaming

----2009 Stephen Somogyi

**Abstract**

Temporal memory streaming replays previously observed miss sequences to eliminate long chains of dependent misses. Spatial memory streaming predicts repetitive data layout patterns within fixed-size memory regions. Because each technique targets a different subset of misses, their effectiveness varies across workloads and each leaves a significant fraction of misses unpredicted.

In this paper, we propose Spatio-Temporal Memory Streaming(STeMS) to exploit the synergy between spatial and temporal streaming. Using trace-driven and cycle-accurate simulation across a suite of commercial workloads, we demonstrate that with similar implementation complexity as temporal streaming, STeMS achieves equal or higher coverage than spatial or temporal memory streaming alone, and improves performance by 31%, 3%, and 18% over stride, spatial, and temporal prediction, respectively.

**1 Introduction**

Larger caches approach provides diminishing returns in today’s multi-megabyte caches and is less appealing in chip multiprocessor (CMPs) where the silicon area can be used instead for additional cores. CMPs themselves do not solve the memory bottlenecks of traditional multi-chip mutiprocessors. While some communication missed remain costly, as do off-chip capacity misses to DRAM. Multi-threading can potentially overlap off-chip memory stalls; however, multithreading is only effective when additional threads are available (which may not be the case in commercial server applications) and does not improve response time.

One approach for reducing the performance impact of off-chip accesses is to prefetch the data：

**Simple stride prefetching**: be effective for sciencitific, desktop and engineering applications

1990: Improving Direct-mapped Cache performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers, Norman P. Jouppi

2000: Predictor-directed Stream Buffer, Timothy Sherwood

**Temporal address-correlating prefetchers:** predict recurring sequences of misses, which arise as applications iterate over data structures, even arbitrarily irregular ones common in commercial workloads. These prefetch exploit the observed order between misses, but cannot prefetch deeply because large-scale data traversals do not repeat perfectly.

1997: Prefetching Using Markov Predictors, Doug Joseph

2002: Using a user-level Memory Thread for Correlation Prefetching, Yan Solihin

2004: Data Cache Prefetching Using a Global History Buffer, Kyle J. Nesbit

2005: Temporal Streaming of Shared Memory, Thomas F. Wenisch

2007: Low-cost Epoch-based Correlation Prefetching for Commercial Applications, Yuan Chou

**Spatial-correlating prefetchers:** predict repetitive spatial layouts over contiguous regions of memory, which arise when applications organize data at a page granularity.

1998: Exploiting Spatial Locality in Data Caches Using Spatial Footprints, Jack L. Lo

2004: Accurate and complexity-effective spatial pattern prediction, Chi F. Chen

2006: Spatial Memory Streaming, Stephen Somogyi

Temporal and spatial prefetchers each target different memory system behaviors; many of the temporally predicted accesses are not predicted spatially, and vice versa. In particular, spatial prefetching suffers from its inability to predict the first miss to each region, and regions are restricted to a fixed size. Temporal prefetching suffers from low accuracy because it does not know where streams terminate and it cannot predict compulsory misses.

**2 Background**

**2.1 Temporal Correlation**

Temporal correlation: sequences of misses are likely to repeat and recent sequences are more likely to repeat than older sequences.

Advantages:

1. The recorded miss sequence includes all misses from a thread, a processor only follows a single sequence when prefetching, as opposed to interleaving misses from multiple sequences.
2. Temporal correlation is ideal for accelerating chains of dependent misses, because sequences contain the miss address themselves, allowing a predictor to fetch the elements of a dependence chain in parallel rather than sequentially.
3. Temporal sequences are frequently long, thus amortizing the startup cost associated with locating/following a new sequence.

Disadvantages:

1. It relies on address repetition, it cannot predict previously unobserved addresses that are common in applications that scan large data sets.
2. Memory addresses can exist as part of many different traversals, and temporal correlation may not be able to identify the best sequence to follow.
3. Training can be slow because a particular code path over a particular data structure must recur before it is predictable.

**2.2 Temporal Memory Streaming**

TMS is a hardware design that exploits temporal correlation .it records the observed miss sequence in a large circular buffer that must be stored in main memory because of its size(~2MB per processor).A key challenge for TMS is locating a particular miss address in the circular buffer, so that it can commence streaming.

TMS has been shown to be effective for OLTP and web serving. One strength of TMS is its ability to parallelize dependent misses that are prevalent in these pointer-chasing workloads, thereby increasing memory-level parallelism. In contrast, TMS is mostly ineffective for DSS workloads, which are dominated by scans of previously untouched data.

**2.3 Spatial Correlation**

Spatial correlation: memory accesses occur in repetitive spatial patterns—that the same offsets, relative to some base address, are accessed.

Advantage:

1. Spatial correlation comes from its use of relative offsets instead of complete addresses.

Disadvantage:

1. An important weakness of spatial correlation is its inability to capture pointer-based dependence: because dynamic objects can be allocated anywhere in the memory space, pointers between two such objects have no inherent spatial relationship.
2. Another shortcoming of spatial correlation stems from its relatively high startup costs.

**2.4 Spatial Memory Streaming**

SMS is a hardware prefetcher that exploits spatial correlation. SMS observes spatial patterns at the L1 data cache and stores them in an on-chip history table for later prediction. A primary challenge in spatial prediction is delineating spatial patterns in both space and time. The second key design chice for spatial prediction is how to associate a spatial pattern with its trigger access.

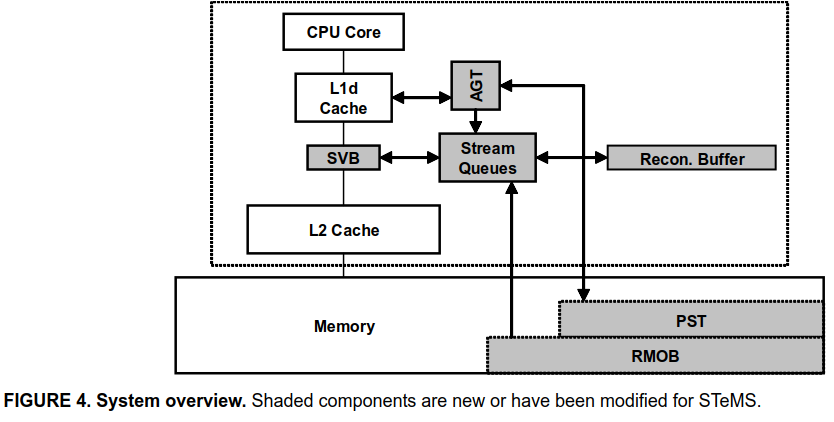
SMS works well for DSS and web serving workloads. DSS queries perform scans over large amounts of data, contained in database pages that all share the same amounts of data, contained in database pages that all share the same layout. Because these pages are traversed by the same code, SMS rapidly learns the spatial access patterns. In contrast, SMS is less effective for OLTP because many accesses that are spatially predictable are already issued in parallel by out-of-order processing.

**3 Spatio-temporal Streaming**

Temporal and spatial correlation are each effective for exploiting different aspects of program behavior. Temporal correlation captures dependence chains and pointers or other sequences that cover large portions of the memory space. In contrast, spatial correlation captures distinct access patterns for different program behaviors over restricts regions of memory. The goal with spatio-temporal streaming is to enable efficient streaming of both temporally and spatially correlated memory behaviors, integrate knowledge of large-scale behavior with small-scale details.

**3.1 Hybrid Spatio-Temporal Prediction**

**4 Hardware Implementation**



**4.1 High-level Operation: training**

**Spatial predictor trains independently**

The predictor observes all L1 access, AGT(Active generation table) accumulates access to each spatial region over the course of a generation, and upon generation termination, PST(Pattern Sequence Table) stores the observed spatial sequence.

**Region misses record in circular buffer**

Much like TMS, STeMS records the temporal miss sequence of (spatial) triggers in an off-chip circular buffer----RMOB(Region Miss Order Buffer). In STeMS, however, misses that are spatially predictable are omitted from the temporal sequence. Hence, prior to appending a miss to the RMOB, STeMS queries the spatial predictor, and only performs the append for trigger accesses and spatial misses. Each PMOB entry contains the block address, the PC of the miss instruction, and the reconstruction delta.

**4.2 High-level Operation: Streaming**

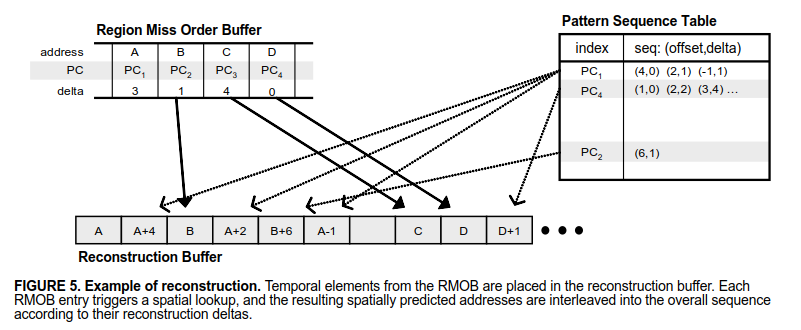
The streaming mechanisms operate very similar to TMS: off-chip misses can iniate new streams and the prefetcher throttles streaming to match application demand. The key difference is that TMS reads the address sequence directly from the circular buffer, while STeMS must reconstruct its prediction sequence.

**Lookup on off-chip miss**

Upon every off-chip miss, STeMS identifies the most recent occurrence of the address in an RMOB and sends subsequent RMOB entries to the requesting processor. As entries are consumed by the reconstruction process, STeMS fetches additional RMOB entries so that reconstruction can resume and the stream can continue as long as possible.

**Reconstruction**

STeMS constructs a total predicted miss sequence using both temporal and spatial predictions.



1. TeMS places the initial miss address at the start of a reconstruction buffer
2. The predictor inserts the addresses from subsequent RMOB entries into the buffer, leaving as many empty spaces as their deltas indicate.
3. STeMS calculaes the spatial lookup index for each RMOB entry using the address and PC, and looks up this index in the pattern sequence table. If found, for each element in the spatial sequence, STeMS calculates the address and inserts it into the reconstruction buffer according to its delta.

**Streaming**

After reconstruction, STeMS moves the sequence of addresses to a stream queue and fetches predicted cache blocks to the requesting processor in order, placing them in a SVB(Streamed Value Buffer). When a block is consumed from the SVB, STeMS fetches the next block according to the stream queue. To reduce erroneously fetched blocks due to invalid streams, only a single block is fetched at the beginning of a new stream. If the block is consumed, the stream is likely to be useful, and further blocks are fetched. When the number of available prefetch addresses in a queue drops below a threshold, STeMS resumes reconstruction from where it left off previously, adding more addresses to the end of the stream queue.

**Spatial-only Streams**

To achieve any coverage on compulsory-miss region, STeMS must support spatial-only streams. During reconstruction, when STeMS queries the spatial predictor for each RMOB entry, the AGT remembers the lookup index for each region. During program execution, as spatial generations begin, the lookup index of the trigger access for each generation is computed and compared to the reconstruction index. If they differ, or if the region was not predicted during reconstruction, STeMS initiates a spatial-only stream using the spatial sequence contained in the PST for the correct index. STeMS treats these spatial-only streams like reconstructed streams, except it ignores the delta information.

**4.3 Hardware Cost**

**Spatial prediction**

A patial sequence requires 32\*10bits=40bytes: for each of 32blocks, 2 bits for the saturating conter vaue and 8 bits for the reconstruction delta. AGR(64 entries) requires 2.5KB of SRAM. With 16K entries, the PST requires 640KB per processor.

**Temporal prediction**

Because some misses are filtered, the overall size of the buffer is reduced from 384K entries(2MB) for TMS to 128K entries(1MB) for STeMS. Each entry contains 16 bits for the PC and 8bits for the delta, 5-byte physical address, totaling of 8B per entry.

**Reconstruction**

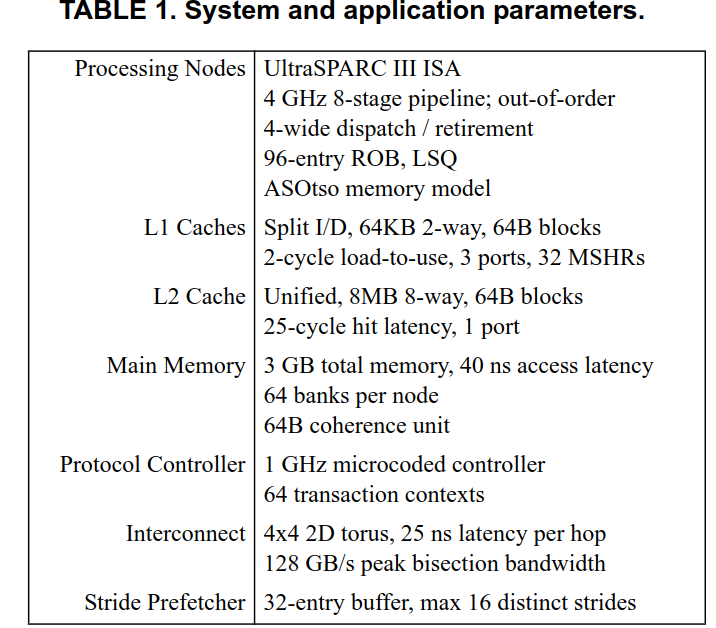
The reconstruction process requires a finite state machine(capable of bit shifts and 10-bit addition) and temporary storage (the reconstruction buffer, 256 entries).

**Streaming**

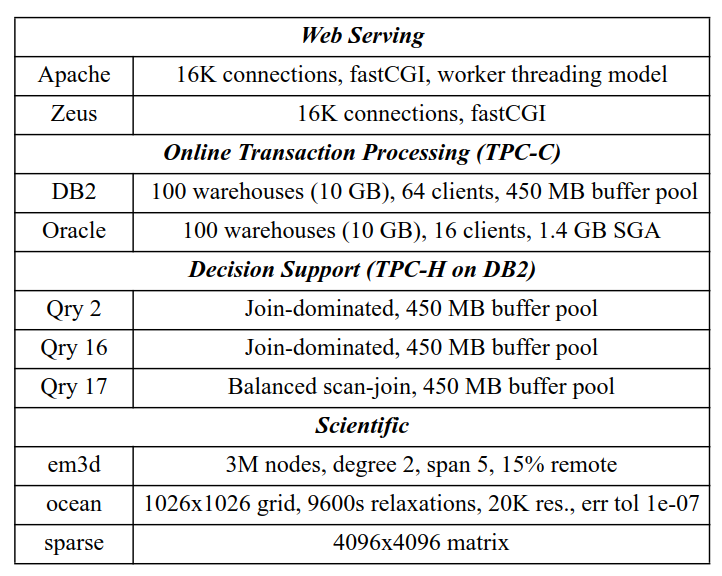
1. **Evaluation**
   1. **Methodology**

**Baseline**

We evaluate STeMS using cycle-accurate full-system simulation of a shared-memory multiprocessor using FLEXUS. Flexus models the SPARC v9 ISA and can execute unmodified commercial applications and operating systems. We simulate a 16-processor directory-based shared-memory multiprocessor system running Solaris 8. WE CONFIGURE OUR PROCESSOR MODEL TO APPROXIMATE THE HARDWARE RESOURCES OF THE Intel Core 2 microarchitecture. We use a store-wait-free memory model to minimize the penalty of stores and memory ordering instructions, thus exposing more off-chip read stalls compared with a conventional TSO system.



**Benchmarks**



* 1. **Comparing Temporal and Spatial Correlation**
  2. **Temporal Correlation Across Spatial Regions**

# 2009: Access Map Pattern Matching Prefetch: Optimization Friendly Method

----2009 Yasuo Ishii

**1 Introduction**

Conventional prefetchers use (a) data address, (b) memory access orderings, and (c) instruction address to generate prefetch requests. However, memory access orderings are often scrambled by out-of-order execution with relaxed memory consistency, and memory access instructions such as load or store are duplicated by loop unrolling. Such scrambles degrade performance of conventional prefetch methods such as GHB.

AMPM(Access Map Pattern Matching) is tolerant to optimizations since it uses only memory access footprint which implies the memory location is recently access or not, instead of the fine-grained memory access sequence information.

AMPM prefetch method involves the following steps:

1. Detecting hot zones
2. Storing the 2-bit states for all cache lines in the memory access pattern map of these hot zones
3. Listing prefetch candidates by pattern matching of the memory access pattern map
4. Selecting prefetch requests from among these candidates and issuing the requests to the main memory

**2 Design of the AMPM Prefetcher**

**2.1 Overview of the AMPM prefetcher**

(1) **Hot zone:** the AMPM prefetcher divides main memory space into fixed sized areas, and detects “hot zones” on the basis of the recent-zone-access frequency.

(2) **Memory access map table:** the AMPM prefetcher stores the 2-bit access states of all cache lines of hot zones in the memory access patterns maps. The number of hot zones and memory access pattern maps is fixed, and they are stored in a memory access map table and replaced by the LRU policy.

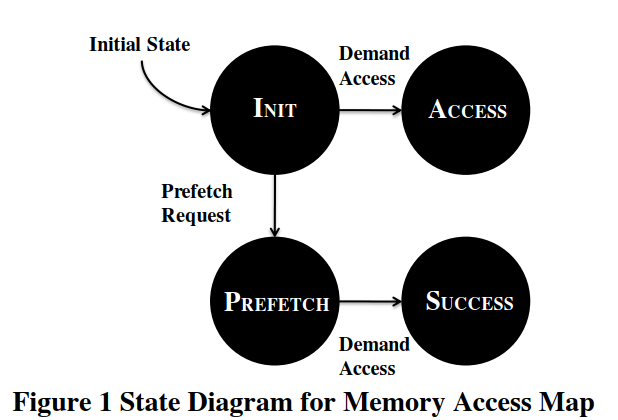
(3) When memory access requests arrive, the AMPM prefetcher tries to detect the stride address correlation by pattern matching using the memory access pattern map and determines the prefetch candidates.

(4) It also decides which prefetch requests are appropriate and issues these requests to the main memory.

(5) The AMPM prefetcher also decides the number of requests to issue on the basis of the profiled information.

**2.2 Memory Access Map**

The entry of memory access pattern map is cache line granularity, the status of each cache line in the zone is stored in a two-bit state machine. This two-bit state machine has four states (Init, prefetch, access, and success).



When almost all states in the zone become access or success, the prefetcher does not issue prefetch request.

**2.3 Generating Prefetch Requests**

The AMPM prefetcher generates the prefetch requests when the L2 cache memory receives a demand request. The prefetcher reads three consecutive memory access maps from the table and concatenates them. The prefetch generator selects the prefetch candidates by determining the address correlation on the basis of the pattern matching in the concatenated map.

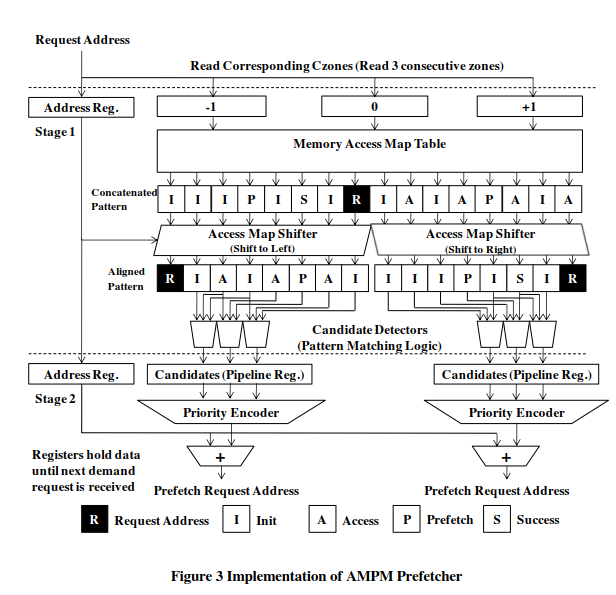
**2.4 Adaptive Prefetch Degree**

The generator controls the prefetch degree using

1. The frequency of the prefetch requests: The AMPM prefetcher employs the access frequency counter and decides its prefetch degree by the quotient of memory access latency and the memory access frequency.
2. The frequency of conflict misses in the L2 cache memory: detected when a memory access map entry of a cache miss address is access or success since such line is not only recently accessed but also evicted from the cache.
3. The conflict misses in the memory access map: detected from the replacement frequency of the memory access map.
4. The ratio of prefetches success: when the number of occurrence of success in the map is large than that of prefetch, the prefetcher uses the prefetch states as access or success. In this case, the prefetch generator can detect more candidates.

**3 Hardware Design & Complexity**

Structure: Memory access map table (CAM) + prefetch generator (two access map shifters, candidate detectors, priority encoders, and address offset adders)



Requests generate steps:

1. The memory access map table is read by the address of the demand request. The shifters are used for memory access map alignment. The position of the demand request is aligned with the edge of the access map.
2. The detector determines candidate address by pattern matching. The priority encoders select prefetch requests to be issued.
3. The addresses corresponding to the issued prefetch requests are calculated in the address offset adders and the requests are issued to the main memory.

**3.1 Memory Access Map Table**

Memory access map table is implemented as a multi ported CAM that holds approximately 64 maps. The complexity of the CAM is almost the same as that of the full-associative TLB, and located in a slower clock domain. This implies that the memory access map table has enough feasibility since it has to operate it has to operate in a slower clock domain. In order to increase the operation frequency, the memory access map table can be implemented as a multi banked set-associative structure, similar to the implementation of the multibanked cache memory.

**3.2 Prefetch Generator**

A prefetch generator is composed of memory access map shifters, candidate detectors, priority encoders, and address offset adders. Each component processes approximately 256bits of data.

**3.3 Pipelining for AMPM prefetcher**

In order to use AMPM prefetch technique at a higher operation frequency, the prefetch generator can be pipelined. Pipeline registers are inserted between the candidate detector and the priority encoders.

**4 Other Optimizations**

**4.1 Processor-Side L1 Prefetching**

Our prefetcher employs an adaptive stream prefetcher as a processor-side prefetcher since an adaptive stream prefetcher has a good cost performance.

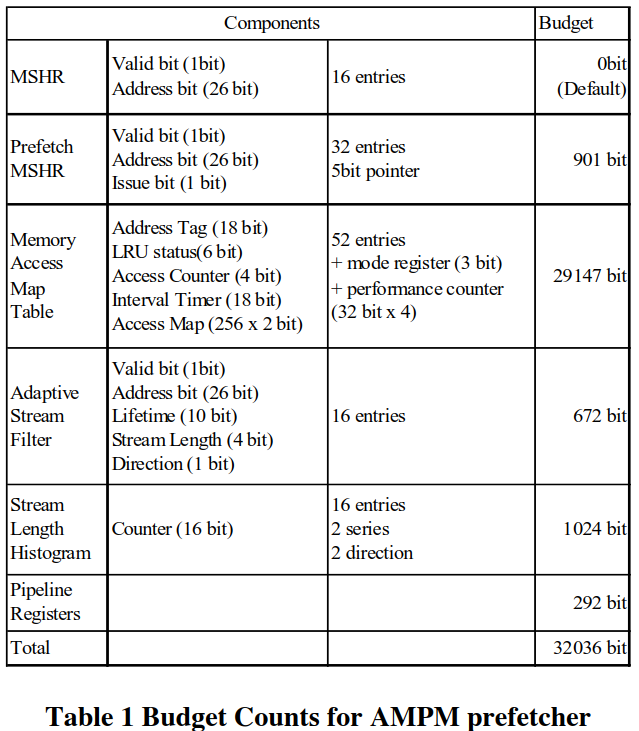
**4.2 Miss Status Handing Register**

The DPC framework provides a MSHR (Miss Status Handling Register) with 16 entries; however, this MSHR size is not sufficient for supporting the required number of in-flight prefetch requests. We employ another MSHR with 32 entries for handling the prefetch requests and use the default MSHR with 16 entries for handling the demand requests.

**5 Evaluations**

**5.1 Configuration**

The prefetcher employs a pipelined AMPM prefetcher and an adaptive stream prefetcher.



* 1. **Results**

The AMPM prefetcher improved performance by 53% and reduced L2 cache miss counts by 74%.

1. **Concluding**

**2014：Sandbox Prefetching**

**Abstract**

Memory latency is a major factor in limiting CPU performance, and prefetching is a well-known method for hiding memory latency.

# 2015: A best-offset Prefetcher

----2015 Pierre Michaud

The BO (Best-Offset) prefetcher submitted to the DPC2 contest prefetches one line into the L2 cache on every cache miss or hit on a prefetched line. The BO prefetcher tries to find automatically an offset value that yields timely prefetches with the highest possible coverage and accuracy. It evaluates an offset value by maintaining a table of recent requests address and by searching these addresses to determine whether the line currently requested would have been prefetched in time with that offset.

**1 Offset prefetching**

Offset prefetching superficially resembles stride prefetching and stream prefetching, but is a more aggressive prefetching methods.

* 1. **sequential stream**

An offset prefetcher yields 100% prefetch coverage and accuracy on sequential streams, like a next-line prefetcher, but can deliver timely prefetchs if the offset is large enough.

Scrambling may degrade prefetch coverage. Some scrambling at the L2 cache may be caused by out-of-order scheduling of instruction and/or L1 miss requests. Some scrambling at the L2 may also happen if there is an L1 prefetcher.

* 1. **Stride stream**

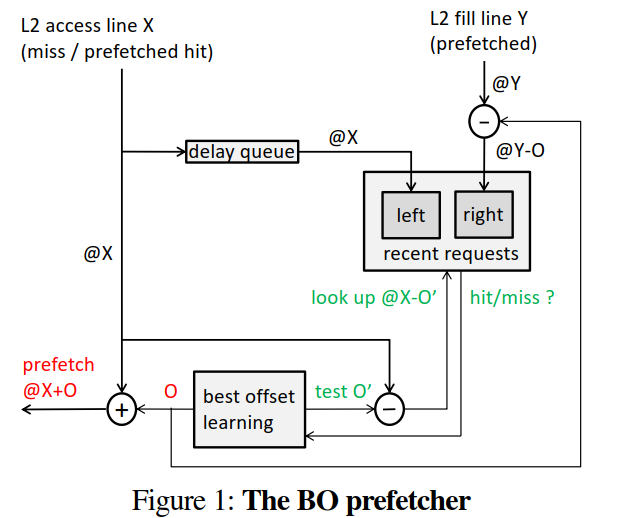
Offset prefetching can (in theory) deliver 100% coverage and accuracy on any periodic sequence of line stride, by setting the offset equal to the sum of the strides in a period, or equal to a multiple of that number.

* 1. **Interleaved stream**

Two interleaved streams S1 and S2 can be prefetched perfectly with a multiple of S1 period multiply S2 period as offset.

1. **The BO prefetcher**

A full-fledge offset prefetcher should have a mechanism for finding automatically the best offset value for each application. The offset could set by using a hardware solution.



1. When reset, best offset O=1, all offset score =0; and all possible offset list is stored in a ROM
2. On every L2 miss or prefetched hit of address x, we issue prefetch x+O
3. On every L2 miss or prefetched hit for a line address x, we test the nth offset On from the list by searching if the line address X-ON is in the RR table. If hit, score for offset 0n is increased. The next L2 access tests offset On+1 and so on.
4. A learning phase finishes at the end of a round when either of the two following events happens first: one of the scores equals SCORE\_MAX, or the number of rounds equals ROUND\_MAX. The best offset will be the prefetch offset during the next phase. Then, the scores and the round counter are reset, and a new phase starts.
5. When a prefetched line Y is entered into the L2 cache (even if that line was demand-accessed after the prefetch request was issued), we write address Y-O into the RR table.

**2.1 RR(Recent Request) table**

Has two bank, each of the two bank has 64entries, and each entry holds a 12-bit tag, two banks are indexed through different hashing function. A hit in the RR table happens when there is a tag match in any of the two banks. A write and read could happen during the same cycle.

**2.2 List of offset**

The list contains 46 offsets (23 positive, 23 negative). The offset list is hard coded in a ROM. Each score is coded on 5 bits.

**2.3 Delay Queue**

There are cases where a small offset gives late prefetches but greater coverage and accuracy.

When a prefetch request X+O is issued, the base address X is enqueued into the delay queue. The delay queue holds address X for a fixed time. After this time has elapsed, address X is dequeued and is wittenn into the “left” bank of the RR table.

1. **Prefetch throttling**

For some applications, prefetching may hurt performance, in particular by polluting the L2 and LLC or by wasting memory bandwidth. Several hardware solving approach:

1. Conservative prefetching
2. Prefetch throttling
3. Try to make useless prefetches harmless

The DPC2 prefetcher can only observe what happens at the L2 cache. In particular, there is no way to directly **measure LLC pollution and memory traffic.** We monitor the best score and LLC access rate for throttling prefetches: dropping prefetch requests when the MSHR occupancy exceeds a threshold, or turning prefetch off completely.

* 1. **Turning prefetch off and on**

If the best score is less than or equal to affixed BAD\_SCORE, we turn prefetch off during the next phase. While prefetch is off, best-offset learning must continue so that prefetch can be turned on if the program behavior changes and the score exceeds BAD\_SCORE.

* 1. **MSHR threshold**

We decrease the MSHR threshold when we detect that memory bandwidth is close to saturation. We use LLC access rate as a proxy for memory bandwidth usage. MSHR threshold increases as LLC access rate increasing.

1. **Related prefetchers**
2. ROT prefetcher[4]

1992: Toward Scalable Cache Only Memory Architectures, E. Hagersten

The ROT prefetcher is a stream prefetcher where each stream may have a different stride. In order to achieve prefetch timeliness, the prefetch distance is increased automatically upon detecting late prefetches. The ROT prefetcher detects streams by associating a score with every stride in a list of stride candidates.

1. Sandbox prefetcher[11]

2014: Sandbox Prefetching: Safe Run-time Evaluation of aggressive prefetchers, S. H. Pugsley

To test an offset, the Sandbox prefetcher dose fake prefetches with that offset by setting corresponding bits in bloom filter. If a subsequent demand request hits on these bits, the fake prefetch is deemed successful.

1. **some simulation results**

For typical sequential streams, the BO prefetcher outperforms the best fixed offset. Streams with a non-unit stride, BO prefetcher produces the same speedup as the best fixed offset.