DSTRIDE: Data-cache miss-address-based stride prefetching scheme for multimedia processors

Hariprakash. G*, Achutharaman. R* and Amos R. Omondi+

*Sun Microsystems,
1 Magazine Road #07-01/13,
Singapore.
{harig,achutha}@singapore.sun.com

+School of Computer Engineering,
N4 Nanyang Avenue,
Nanyang Technological University, Singapore.
ASAmos@ntu.edu.sg

Abstract

Prefetching reduces cache miss latency by moving data up in memory hierarchy before they are actually needed. Recent hardware-based stride prefetching techniques mostly rely on the processor pipeline information (e.g. program counter and branch prediction table) for prediction. Continuing developments in processor microarchitecture drastically change core pipeline design and require that existing hardware-based stride prefetching techniques be adapted to the evolving new processor architectures.

In this paper we present a new hardware-based stride prefetching technique, called DStride, that is independent of processor pipeline design changes. In this new design, the first-level data cache miss address stream is used for the stride prediction. The miss addresses are separated into load stream and store stream to increase the efficiency of the predictor. They are checked separately against the recent miss address stream to detect the strides. The detected steady strides are maintained in a table that also performs look-ahead stride prefetching when the processor stride reference rate is higher than the prefetch request service rate.

We evaluated our design with multimedia workloads using execution-driven simulation with SimpleScalar toolset. Our experiments show that DStride is very effective in reducing overall pipeline stalls due to cache miss latency, especially for stride-intensive applications such as multimedia workloads.

1. Introduction

Multimedia workloads are very memory intensive and usually generate many cache misses when run. For example, motion compensation and audio or video compression are heavily memory bounded; these programs spend most of their runtimes stalled on memory requests. Prefetching has long been known to significantly decrease the cache miss latency. Increase in demands for processing power and corresponding advancements in processor pipeline design have lead to the emergence of multimedia processors and general-purpose processors enhanced with multimedia extensions, but more careful study is needed in order to adopt any of the existing hardware based stride prefetching techniques to the emerging microarchitectures.

In this paper, we propose a new design for a hardware based stride prefetching scheme which is intended to

- operate independent of processor pipeline design changes,
- reduce the cache miss latency, especially for multi-media applications.

Our design of stride prefetching uses the miss reference stream and so is different from existing program counter based stride prefetching schemes [1, 6, 15]. Our predictor maintains a Stride Prediction Table (SPT) that compares each recent miss address with several previous miss addresses to calculate all possible strides. State bits are maintained to identify steady strides. A *Steady Stride Table* (SST) maintains the steady strides detected by the SPT and issues prefetch requests. We maintain a separate prefetch buffer cache to avoid cache pollution that may occur because of prefetched data replacing useful cache blocks.

We evaluated our design using three metrics: the Memory Cycles Per Instruction (MCPI), which is the number of memory cycles per instruction; Relative MCPI (RMCPI), which is the MCPI relative to the baseline system configuration without prefetching; and the Global Success Ratio, which is the fraction of cache misses which are avoided by the prefetching [4]. On average, for multimedia workloads, our design reduces the MCPI contribution by about 60-75% and the GSR is increased by about 50-75% when compared to the baseline system configuration.

The outline of the rest of the paper is as follows. In Section 2, we discuss related work done in hardware-based prefetching. Section 3 describes the detailed design and implementation of our prefetcher, DStride. In Section 4, we discuss our simulation environment and the performance evaluation of DStride. Section 5 consists of a summary with a discussion of future work.

2. Related work in hardware-based prefetching

Hardware-based prefetching can be classified into two categories: program counter (PC) based scheme and data-address based schemes [9]. PC-based scheme operates on reference address stream, by using the PC value and branch prediction table to determine whether to prefetch and to initiate any subsequent prefetch request. On the other hand data-address-based schemes operate mostly on reference miss address stream and relies on the data address itself to decide when to initiate a prefetch request.

2.1. PC-based prefetching schemes

Baer and Chen [1] investigate a hardware data prefetching scheme that uses a Reference Prediction Table (RPT) and Look-Ahead Program Counter (LA-PC). The RPT is a cache whose tag field contains the address of a load/store instruction and whose data field contains the last address referenced by that instruction and the corresponding stride. The LA-PC is a secondary PC that is used to predict the execution stream in situations where the loop iteration time is smaller than the memory latency. The LA-PC is modified via the Branch Prediction Table (BPT). When the LA-PC value is that of a load/store instruction, the next address is calculated by adding the address with the stride of that RPT entry and a prefetch is issued.

The original Baer-and-Chen prefetching scheme cannot work with superscalar processors, but Pinter and Yoaz in their Tango have managed to modify it appropriately [6]. The major difference in Tango with respect to Baer and Chen are: 1) The LA-PC and RPT are difficult to adapt to multiple issue environment. Tango defines a Pre-PC in order to adapt to multiple-issue processors. 2) In contrast to the LA-PC, the Pre-PC look-ahead scheme in Tango scans only the branches and the memory access instructions. This is done by Pre-PC using a extended Branch Target Buffer (BTB) with additional Program Process Graph (PPG).

The major disadvantage of PC-based schemes is that they very much depend on the pipeline design. There are now several new microarchitectures, for example, speculative execution, multiscalar, multithreaded etc., and PC based schemes to these architectures have yet to be adapted to these. We think data-address-based schemes are superior to PC based schemes because of their inherent advantage of not depending on the underlying pipeline microarchitecture.

2.2. Data-address-based prefetching schemes

2.2.1. Cache prefetcher. Generally a cache satisfies processor references by demand fetching. Cache prefetching is the loading of a block before it is referenced by the pipeline [4]. There are various cache prefetching policies: One Block Look-ahead (OBL) policy, in which upon referencing a block i, the next block, i+1, is prefetched; prefetch on miss; prefetch unconditionally; prefetch on previous successful prefetch; and so forth. The drawback of the cache prefetching schemes is that useless prefetches can pollute the cache by displacing the useful cache blocks from the cache. The other disadvantage is that none of the cache prefetching policies above is suitable for "stridy" workloads.

2.2.2. Stream buffer. Jouppi introduced Stream Buffers as a load/store latency-reduction technique [10]. The Stream Buffer proposed by Jouppi is a FIFO stream buffer that prefetches a sequential stream of cache lines starting at a given address. The main drawback of the FIFO Stream Buffer is not capable of handling non-unit strides.

Palacharla and Kessler [11] modified FIFO stream buffers to handle non-unit stride detection, by dynamically partitioning the physical address space and detecting the strides within the partition. In their scheme each physical address issued by the pipeline is divided into a *tag* (higher order bits) and a *czone* (concentration zone). The *czone* is set at runtime. Two references will fall within the same partition if their addresses have the same tag bits. A history buffer (non-unit stride filter) and a finite state machine are then used to detect non-unit strides. At the end of three consecutive strided references, a stream is allocated and the entry in the history buffer is freed. A unit-stride filter is also used.

Setting the *czone* bit at runtime requires a software bit-mask that must be individually adjusted for a given application and architecture [11, 13]. This seems to be impractical for even a small set of multimedia workloads [14]. Our stride detection logic, discussed in Section 3.3, can be used with stream buffers to detect both unit and non-unit strides but does not have these drawbacks.

2.2.3. Markov predictor. A Markov predictor uses the miss-address stream as the prediction source [12]. The Markov model relies on past references to predict future references when a past reference is repeated. A hardware approximation to the Morkov model maintains a prefetch table with several past miss addresses and several possible subsequent references for each miss address. When the current miss address matches any of the miss addresses in the prefetch table, all of the next-reference addresses associated with the address are eligible for prefetch. The Morkov predictor works well for the programs in which the same address patterns are repeated and its performance for instruction prefetching is better when compared to data prefetching - but it does not handle stride prefetching.

3. Design of DStride prefetcher

3.1. Motivation

New developments in microarchitecture significantly change the design of processor pipeline, so any

hardware-based stride prediction technique which closely interacts with pipeline must be modified for every major change in microarchitecture. For example, Bear and Chen's [1] PC-based prefetching scheme for scalar processor was redesigned by Pinter and Yoaz [6] in their Tango implementation to adopt it for superscalar (multiple issue) processors. However, PC-based stride prediction schemes can only identify strides that are enclosed in loop. Data-address based prefetching schemes detect both strides that are enclosed in loop, but also detects those that are not.

As an example, consider the code fragment below. This is taken from MPEG2 code [7] and is used in `idctcol()`, which is a heavily used function in MPEG Encode/Decode programs to calculate the two-dimensional inverse discrete cosine transform. The access to `blk[8*i]` forms the a stride access, and each access causes a new cache line to be fetched for store operation in a typical 16-byte block of a cache.

```
static void idctcol(blk)
short *blk;
{
...
/* fourth stage */
blk[8*0] = iclp[(x7+x1)>>14];
blk[8*1] = iclp[(x3+x2)>>14];
blk[8*2] = iclp[(x0+x4)>>14];
blk[8*3] = iclp[(x8+x6)>>14];
blk[8*4] = iclp[(x8-x6)>>14];
blk[8*5] = iclp[(x0-x4)>>14];
blk[8*6] = iclp[(x3-x2)>>14];
blk[8*7] = iclp[(x7-x1)>>14];
}
```

A data-address-based prefetching scheme can detect the stride in this code and issue a prefetch when the third reference to the `blk[]` is issued. But a PC-based stride predictors, since it relies on the PC to identify the stride when the loop repeats, cannot detect the stride because it is not enclosed in a loop. The above code fragment has unit stride, so a stream-buffer logic seems to be sufficient to handle the prefetches. However, non-unit strides cannot be handled by stream buffers without additional non-unit stride detection logic to detect them. We next present the design of an appropriate stride detector.

3.2. Basic block diagram

DStride, the new hardware prefetcher that we propose, consists of a Prefetch Buffer, Prediction Logic, and Prefetching Logic, arranged as shown in Figure 1; In a typical realization, all of the logic shown, except for the Level-2 cache, will be on-chip.

The Prefetch Buffer (PB) holds the predicted data, which are prefetched ahead in time. We opted to have a separate on-chip Prefetch Buffer to store the prefetched data in order to avoid cache pollution in the data-cache; the buffer is logically at the same level as the data cache. Both the data cache and Prefetch Buffer are searched in parallel for a match with reference addresses coming from the pipeline. On an address match in the prefetch buffer, the data is passed to the pipeline and the address is passed to the prefetching logic to prefetch according to any predicted strides.

The Prediction Logic sits below the data cache and uses the data cache miss address stream to predict the future references [2, 5]. The predictor consists of Stride Prediction Tables (SPT) that detect steady strides, by comparing the current miss address with recent miss addresses. The SPTs have associated adder/subtracter units to calculate the strides.

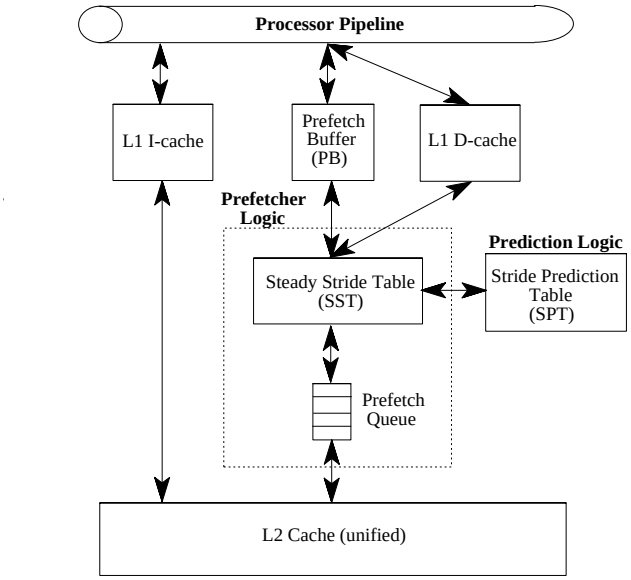

Figure 1.Schematic block diagram of DStride

The Prefetching Logic consists of a Steady Stride Table (SST) and a Prefetch Queue (PQ). When the SPT detects a steady stride, the corresponding entry is removed from SPT and moved into SST. When a data-cache miss address or a Prefetch Buffer hit address matches with a SST entry, a prefetch request is issued for the next stride address. The prefetch requests are queued in the PQ, which acts as an interface between the on-chip SST and the off-chip level-two cache.

3.3. Prediction logic

A data-address based hardware predictor predicts future references based on previous access patterns by using either the address reference stream (sequence of addresses referenced by the processor pipeline) or the miss address stream (the data-cache miss addresses).

In the DStride predictor, the miss address stream is used to detect the strides. Miss addresses fed into Prediction Logic are aligned to the data-cache block size and so the predicted strides are of multiples of data-cache block size. In the case of strides that which are smaller than the data-cache block size, accesses are to the same cache line. When the smaller strides cross the cache block boundary, they cause cache misses, which are tracked by

the prediction logic in the miss address stream in order to correlate and identify the strides.

3.3.1. Stride Prediction Table. The main component of the prediction logic is the Stride Prediction Table (SPT). The predictor uses two different SPTs to track the load and store addresses separately (Load SPT and Store SPT) to increase the efficiency of the predictor. The rationale for the separate SPTs is given below. The SPT records recent miss addresses and compares the current input address with the previous addresses to calculate the strides between them. In other words, the SPT calculates all possible strides for an address by comparing it with a set of recent addresses. From the comparison, SPT can calculate several different strides. These addresses forms a window for comparison to detect the strides. For example, consider the loop

```
int a[1024],b[1024],c[1024];
register int i,j,k,l;

for(l=0;l<1024;l++){
        a[i] = 10; i +=10;
        b[j] = 20; j +=20;
        c[k] = 30; k +=30;
}
```

The array `a,b` and `c` are accessed after every three memory references. For the SPT to detect the stride of each array, the window of comparison should be at least 3. The schematic block diagram of a DStride predictor with a 4-window SPT is shown in Figure 2.

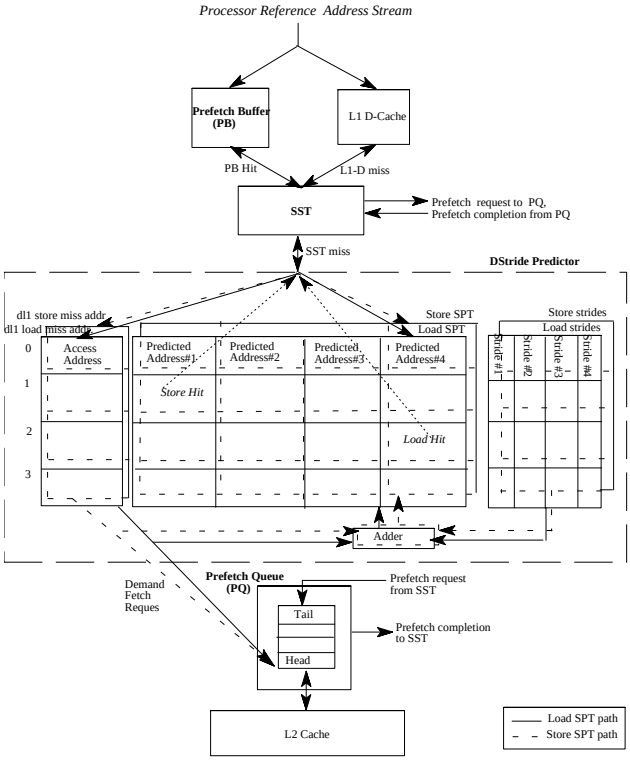

Figure 2. DStride with 4-window SPT hardware block diagram

An SPT entry consists of four fields:

- *miss address*, which holds a data-cache miss address;
- *strides*, which holds a fixed number of strides;
- *predicted addresses* - one next-address for each stride, based on the current miss address and the stride;
- two *state bits* that partially encode the past history and are used to determine the steady state for prediction.

The SPT stride calculation is explained with an example of a 4-window SPT, shown in Table 1.

Table 1. Four-window Stride Prediction Table

Miss addresses	Strides				Predicted Addresses			
	S0	S1	S2	S3	Pa0	Pa1	Pa2	Pa3
d0	(d1-d0)	(d2-d0)	(d3-d0)	(d0'-d0)	d1+S(0,0)	d2+S(1,0)	d3+S(2,0)	d0'+S(3,0)
d1	(d2-d1)	(d3-d1)	(d0'-d1)	(d1'-d1)	d2+S(0,1)	d3+S(1,1)	d0'+S(2,1)	d1'+S(3,1)
d2	(d3-d2)	(d0'-d2)	(d1'-d2)	(d2'-d2)	d3+S(0,,2)	d0'+S(1,2)	d1'+S(2,2)	d2'+S(3,2)
d3	(d0'-d3)	(d1'-d3)	(d2'-d3)	(d3'-d3)	d0'+S(0,3)	d1'+S(1,3)	d2'+S(2,3)	d3'+S(3,3)

Let d_n be the current miss address, d_x be the xth recent miss address against which comparison is made, and d_x' be the current miss address after wrap around of the window and N be the SPT window size. If the nth stride corresponding to d_x is s_{xy} ,
where

$$y = [(N-1)+(n-x)] \text{ modulo } N$$

then

$$s_{xy} = d_n - d_x$$

And if the nth predicted address corresponding to d_x is pa_{xy}, then

$$pa_{xy} = d_n + s_{xy}$$

A set of adders/subtracters that operate in parallel are used to carry out the above calculations for all the recent miss addresses d_x, where $0 <= x < N$, within the window N. When the window is filled, the miss address values are stored by wrapping around in a circular fashion. Similarly, the *strides* and *predicted addresses* wrap around in a circular fashion. When n equals N, the miss address wraps around and occupies the 0th entry and n becomes 0.

Every miss address is associatively searched for among the *predicted addresses* in the Load SPT and the Store SPT. On a match, the corresponding *state bits* are updated and the next predicted address is calculated and stored; on a miss, several strides are calculated as explained above.

3.3.2. SPT states. *State bits* are used to improve the accuracy of the prediction by filtering out irregular and false strides. Two bits are maintained for each predicted address and are a partial encoding of the history. The encoding in these two bits direct future actions on prefetching.

The four states defined by the two bits are

- S_INIT, the initial state, which is set when a *predicted address* is added
- S_TRANS, the transient state, which indicates when the prediction logic is not sure of whether the prediction is correct or not
- S_STEADY, the steady state, which is when the prediction logic determines that the prediction is correct and the stride could be consistent for a while
- S_NOSTATE, which indicates that the entry is free.

When a data-cache miss address matches a *predicted address*, one of two state transitions occurs :

- If the current state is S_INIT, then the state is changed to S_TRANS.
- If the current state is S_TRANS, then the state is changed to S_STEADY and the SPT entry is moved to the SST. After moving the entry to SST, the state is changed to S_NOSTATE to free the SPT entry.

3.3.3. Load SPT and Store SPT. Programs whose access patterns are stridy in nature seem to have sequence of Load or Store accesses to memory. Consider for example, the code fragment

```
char *memcpy(b,a,n)
char *b;
register char *a;
{
register char *d = b;
        while(n--)
                *d++ = *a++;
        return(b);
}
```

There are two streams of accesses in this code: the *Load Stream* is formed by the sequence of Load instructions that are issued to access the pointer `a`; and the *Store Stream* is formed by the sequence of Store instructions that are issued to access the pointer `d`. The loop issues alternate accesses to the two streams. If the SPT is unified, it compares the current access address with several recent addresses to identify any strides, alternate accesses to Load Stream and Store Stream cause the SPT to compare the Load Stream addresses with the Store Stream addresses, which results in useless strides and pollution of the SPT. In PC-based stride predictors, alternate access to the Load and Store streams is automatically taken care of, since only one instruction can be tagged with the PC and it cannot be both a Load and a Store. In the DStride predictor, the pollution is avoided by maintaining two separate SPTs - one for the Load stream and one for the Store stream. The Load SPT compares the current Load miss address with only the recent Load miss addresses and tries to identify a steady stride. Similarly, the Store SPT compares the current Store miss address with only the recent Store miss addresses to identify a steady stride. Thus, the SPT pollution is avoided, and the efficiency of the predictor is significantly improved.

3.4. Prefetching Logic

The Prefetching Logic issues prefetch requests to the next level of the memory hierarchy. The logic consists of the Steady Stride Table (SST), which maintains the steady strides and queues the prefetch requests to Prefetch Queue (PQ).

3.4.1. Steady Stride Table. Steady strides detected by SPT are moved into the SST, which maintains an entry for every such stride. An SST entry and its associated logic are shown in Figure 3. The SST logically sits between the level-one caches (the data-cache and the Prefetch Buffer) and level-two cache. The main function of the SST is to handle look-ahead prefetch. When the processor stride reference rate is higher than the prefetch request service rate, the prefetched data may not be available in time for use. In order to overcome this problem, the SST does prefetching in advance by increasing the look-ahead distance.

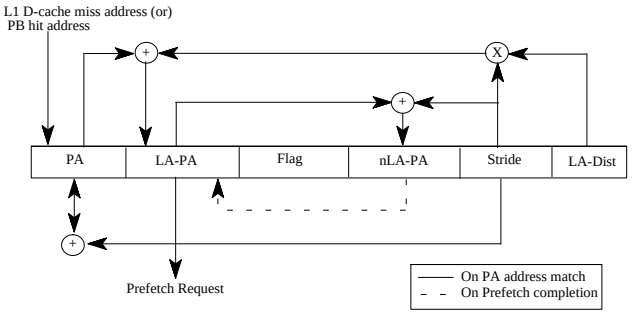

Figure 3. An entry of a Stredy Stride Table

The SST entry has six fields :

- the Predicted Address (PA)
- a Look-ahead Predicted Address (LA-PA)
- a *Flag* that indicates prefetch state information, as
  IDLE - no outstanding prefetch
  PENDING - current prefetch is pending
  PF_PENDING - previous prefetch is pending
- a next Look-Ahead Predicted Address (nLA-PA)
- *Stride*, which is the stride value
- the Look-ahead stride Distance (LA-Dist)

PA and *Stride* are initialized from the SPT when a steady stride is identified by the SPT. Initially the *LA-Dist* is set to 1. *LA-PA* is initialized to the *PA* value and a prefetch request is issued to the PQ. *Flag* is set to *PENDING* because a prefetch request has been issued. *nLA-PA* is calculated as *LA-PA + Stride* and stored. On the completion, of a prefetch the *PENDING* flag is cleared and set to *IDLE* , and the *nLA-PA* value is moved to *LA-PA*.

There are two input streams to SST: the data-cache miss stream and prefetch-buffer hit stream. The input address is associatively searched for in the PA entries of the

SST. On a miss, the input address is passed to the prediction logic (SPT). On a match, *PA* is advanced to the next stride value. The prefetch is issued for the next stride address only when the flag is in *IDLE* state. In this case, the next stride address is available in the *LA-PA*. The prefetch is issued for *LA-PA* value, the flag is set to *PENDING*, and *nLA-PA* is advanced to *LA-PA + Stride.* On prefetch completion, the *PENDING* flag is cleared and set to *IDLE*, and *nLA-PA* value is moved to *LA-PA*.

When the flag is in *PENDING* state, it indicates that an outstanding prefetch is in progress. In this case, the input address is also compared with *LA-PA* to see if the input address itself corresponds to a pending request. If the input address is the same as the *LA-PA*, then it means that *PA* has caught up with *LA-PA*. This scenario can happen when the rate at which the stride is accessed by the pipeline is higher than the rate at which the prefetch requests are serviced and it can happen more easily with shorter loops and in cases where the strides accesses are issued quickly. In order to make the data available earlier than it is actually needed, *LA-Dist* is incremented every time in the powers of two, but, it is not incremented beyond *LA-Limit* in order to avoid unnecessary look-ahead prefetches. The new look-ahead address is calculated as *LA-PA = PA + (Stride * LA-Dist)*. A prefetch is issued on the recalculated *LA-PA* , and the flag is set to *PENDING*. *nLA-PA* is advanced to *LA-PA + Stride*. On prefetch completion the *PENDING* flag is cleared and set to *IDLE*, and *nLA-PA* value is moved to *LA-PA*.

When the flag is in *PENDING* state and the input address does not match the *LA-PA*, it means that the previous prefect request is not yet completed. The flag is set to *PF_PENDING* to notify the SST to issue a prefetch on completion of the previous pending prefetch. On such a completion, *nLA-PA* is moved to *LA-PA* and a prefetch is issued, *PF_PENDING* is cleared, and *PENDING* is set.

To improve the prediction accuracy, the prefetch requests are not issued for data that is already present in the data-cache or in the Prefetch Buffer. When the prefetch address raises a trap for address translation, or when an out-of-range address is issued by the prefetching logic, the prefect request is discarded. A pseudo-LRU policy is used for replacing SST entries.

3.4.2. Prefetch Queue. Prefetch requests and demand-fetch requests (data-cache misses) are queued in the Prefetch Queue (PQ). Demand-fetch requests get priority over prefetch requests. The PQ is located just before the level-two cache and acts as a data router when the data arrives back from that cache. The PQ maintains a table of requests to be serviced by the level-two cache. Each entry of the PQ has the format shown in Figure 4.

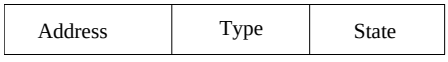

Figure 4. An entry of a Prefetch Queue

The entry consists of three fields :

- *Address*, which is the prefetch/demand fetch request address.
- *Type*, which indicates the type of requests as
  D - demand fetch request.
  P - prefetch request.
- *State*, which indicates the state of a request - *IDLE* or *PENDING*

Request addresses are associatively searched for in the *Address* field of the PQ. If a demand fetch request is already entered in the PQ as a prefetch request, then the prefetch request is converted into a demand fetch by changing the *type* from *P* to *D* and SST is notified to clear the *PENDING* flag for this request. Duplicate requests are not queued and are simply discarded. When the request is selected for service by the level-two cache, the *state* is set to *PENDING*. On completion of the request, the data is routed to the data-cache or the Prefetch Buffer, according to the *type* of the request and the *PENDING state* is cleared to *IDLE*.

4. Simulation study and performance

We used Wisconsin SimpleScalar version-2.0 toolset [3]; this is an execution driven superscalar processor simulator with out-of-order issue capability. We integrated our DStride prefetcher into the SimpleScalar and studied the behavior with MediaBench workloads [14] and with a small set of commonly used real programs. The workload characteristics are summarized in Table 2a and Table 2b.

Table 2a. MediaBench workload characteristics

MediaBench Multimedia workloads	Description	Memory references	Load%	Store%	No. of Executed instruction	Branch%
MPEG2encode	Standard for high quality digital video transmission uses DCT for coding, IDCT for decoding.	924603897	92.6	7.4	1946374852	16.0
MPEG2decode		82275834	63.4	36.6	75854342	11.5
EPIC coder (epic)	Efficient Pyramid Image Compression - bi-orthogonal pryamid transform coder, decoder	7945206	89.4	10.6	55305205	15.0
EPIC decoder (unepic)		1794700	56.1	43.9	7659981	21.1
Mesa -mipmap[1]	Clone OpenGL 3-D graphics library. [1] executes texture mapping, [2] executes rendering pipeline, [3] generates texture mapping	23676377	67.1	32.9	74987642	16.6
Mesa -osdemo[2]		7873475	69.2	30.8	24930922	18.3
Mesa -texgen[3]		32203190	65.4	34.6	105477995	12.9
G.721-encode	CCITT Voice compression	54764749	77.2	22.8	319264177	23.0
G.721-decode		55103707	76.2	23.8	307575444	23.0

Table 2b. General application workload characteristics

General Applications	Memory references	Load%	Store%	No. of Executed instruction	Branch%
memcpy-10k	75414	68.4	31.6	252120	8.0
matmul 100x100	16074334	87.5	12.5	80179377	2.5
kernel6 (livermore loop)	73628	81.8	18.2	280676	3.9
gzip	9956337	70.3	29.7	35608364	18.3
gunzip	2126244	84.9	15.1	7667259	17.9

We studied the performance of DStride prefetcher with various cacheline sizes, cache sizes and associativity by comparing with a baseline system. The default system configuration of the DStride and the baseline system are given in the Table 3.

Table 3. Default system configuration

System parameters		Default Values
Baseline	**DStride**	
Level-1 D-cache	Level-1 D-cache	Size - 16K, Cacheline - 32 bytes, Associativity - 1, Dual ported Tag, Write Allocate, PIPT.
Level-2 cache	Level-2 cache	Size - 4MB, Cacheline - 64 bytes, Associativity - 1, Write Back, PIPT.
mem-bus width	mem-bus width	8 bytes
—	SPT	8 Window - Load/Store SPT
—	SST	256 Steady Stride entries
—	Prefetch Buffer	1024 cache lines
—	Prefetch Queue	4 entries
—	LA-Limit	16 strides

4.1. Evaluation Metric

The studies of J. Tse and A. J. Smith in [4] clearly showed, that a decrease in the cache miss ratio attributable to prefetching does not necessarily actually lead to an improvement in CPU performance. Even when the miss ratio decreases, prefetching can degrade performance because of prefetch lookups on busy cache address tag arrays and a busy memory bus on prefetch address transfers and data fetches. We consider three important metrics taken from [4] to evaluate our design - Cycles Per Instruction contributed by Memory access (MCPI), Relative MCPI (RMCPI) and Global Success Rate (GSR). MCPI is defined as

$$MCPI = \frac{\textit{total memory access penalty}}{\textit{total no. of instructions executed}}$$

The *total memory access penalty* is a sum of data-cache miss penalty and partial data-cache miss penalty - which is the portion of the penalty for those miss requests that were already requested by the prefetcher (due to late prefetch).

We use Relative MCPI to compare the results of DStride prefetching scheme with the same system without prefetching. The RMCPI for the baseline system (without prefetching) is 1. RMCPI is defined as

$$RMCPI = \frac{\textit{MCPI with prefetching}}{\textit{MCPI without prefetching in the same system}}$$

To evaluate the accuracy of our model, we chose Global Success Ratio (GSR) as a metric. The GSR is the fraction of cache misses which are completely avoided or partially avoided (i.e., prefetches already in progress). The GSR is defined as

$$GSR = \frac{\textit{Total number of correct prefetches}}{\textit{(Total number of correct prefetches + Total number of true cache misses)}}$$

A GSR of zero implies that a prefetching strategy does not save any misses, while a GSR of one implies all the cache misses are avoided. The GSR for the baseline system is 0. The goal is to reduce the MCPI, RMCPI and to increase the GSR with data prefetching using DStride model.

4.2. Performance Evaluation

We studied the effect of DStride prefetching statistics by varying the cache block sizes, cache sizes and associativity of the data-cache. We chose DStride's main parameter, the SPT size as 8 entries; smaller SPT size, such as four, will not be sufficient to be the prediction window as it cannot cover the strides that of greater than four. On the other hand, a prediction window of 16 entries does not seems to be cost effective. We chose 8 entry SPT as this seems to provide good coverage and allows a cost-effective implementation.

4.2.1. Effect of cache block size on prefetching. In Figures 5a and 5b, the MCPI is plotted as a function of cache block size for general applications and MediaBench applications.

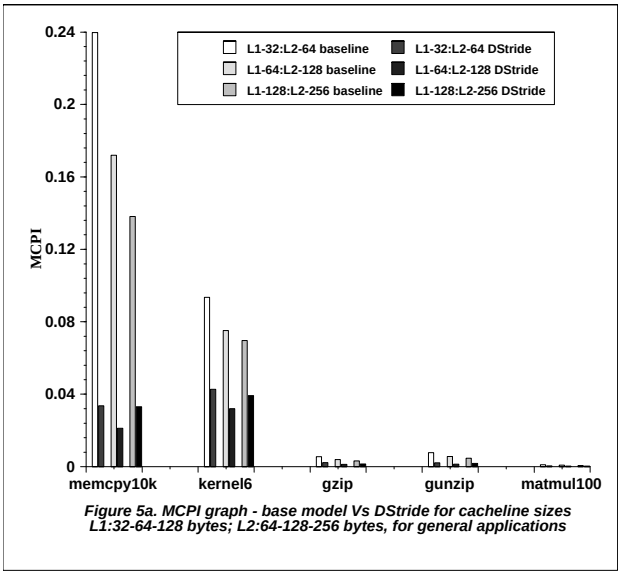

Figure 5a. MCPI graph - base model Vs DStride for cacheline sizes L1:32-64-128 bytes; L2:64-128-256 bytes, for general applications

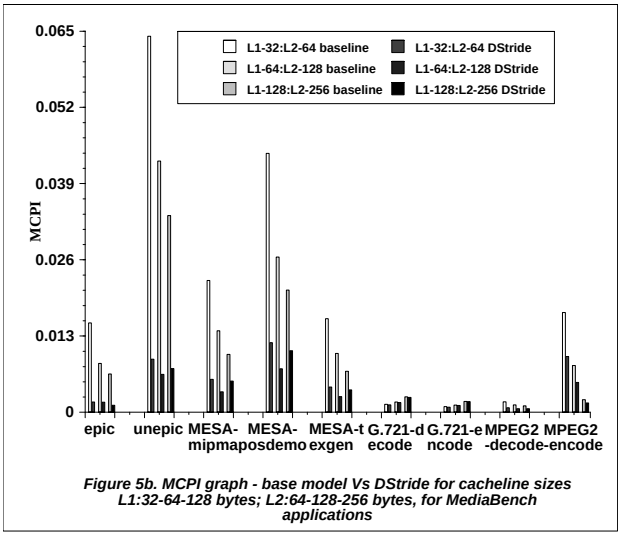

Figure 5b. MCPI graph - base model Vs DStride for cacheline sizes L1:32-64-128 bytes; L2:64-128-256 bytes, for MediaBench applications

On average for, MediaBench video applications, DStride reduces the MCPI up to 75%, for general applications 71% gain in MCPI is obtained. We observe when cache blocks are 32 bytes long, DStride performs better for MediaBench applications, whereas general applications perform better when cache block size is 64 byte long. When cache block size increases, more cycles are required to load a prefetched block; because of the constant bus width of 8 bytes used in our simulation. Consequently, most of the prefetch requests in progress are converted to partial miss when the stride reference rate is higher than the prefetch load latency. This phenomenon is observed as increase in RMCPI in all the applications for 128 bytes level1-cache with 256 bytes level-two-cache shown in Figures 5a, 5c and Figures 5b, 5d, except *epic*. G.721 encoder and decoder have insignificant stride patterns, this is the reason for the RMCPI of these programs are close to that of the baseline system.

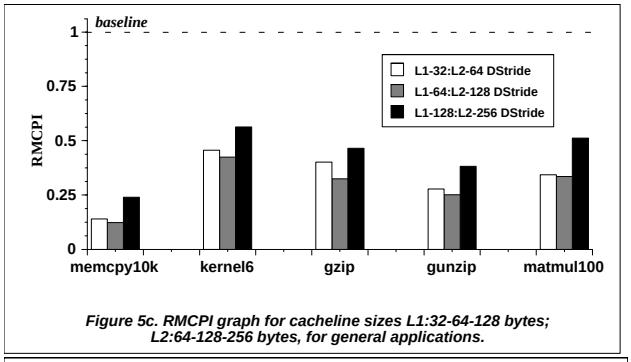

Figure 5c. RMCPI graph for cacheline sizes L1:32-64-128 bytes; L2:64-128-256 bytes, for general applications.

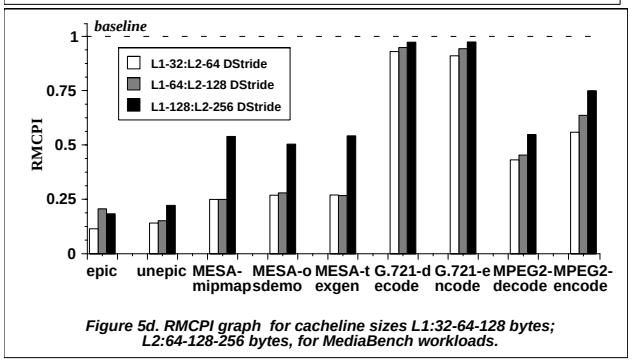

Figure 5d. RMCPI graph for cacheline sizes L1:32-64-128 bytes; L2:64-128-256 bytes, for MediaBench workloads.

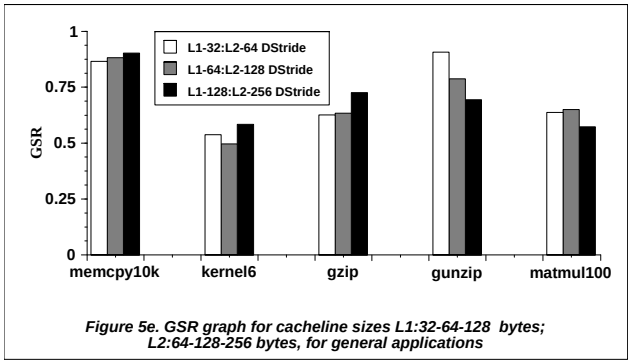

Figure 5e. GSR graph for cacheline sizes L1:32-64-128 bytes; L2:64-128-256 bytes, for general applications

For general applications, we observe that an average GSR is about 75% (Figure 5e), and for MediaBench applications it is about 50% (Figure 5f).

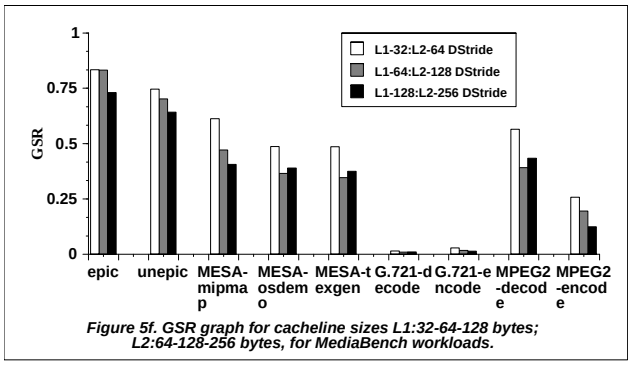

Figure 5f. GSR graph for cacheline sizes L1:32-64-128 bytes; L2:64-128-256 bytes, for MediaBench workloads.

4.2.2. Effect of cache size on prefetching. The results in Figure 6a show that there is no significant gain in RMCPI or GSR as compared to that of 16k cache size with larger size caches. DStride with 16k cache size in all the MediaBench applications shows best performance. The MPEG2, EPIC image encoders and the Mesa programs have many more memory accesses (Table 2a). So we should expect that increase in cache size will have more of an impact on them. This is indeed the case as shown in Figure 6a: as the cache size increases most memory accesses are hits and there are few misses and hence fewer prefetches (as prefetches are done on the miss stream); thus the RMCPI increases. The average RMCPI gain for most of the multimedia workloads is about 60% in almost all the cache sizes (Figure 6a).

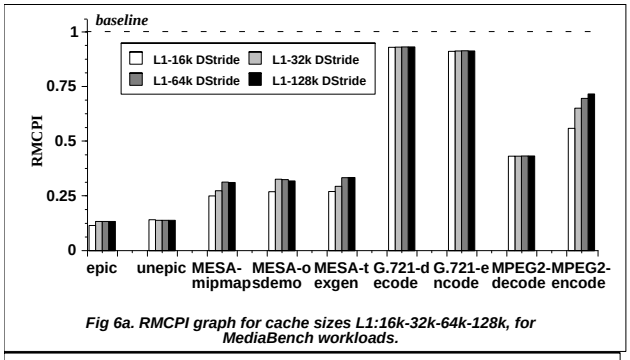

Fig 6a. RMCPI graph for cache sizes L1:16k-32k-64k-128k, for MediaBench workloads.

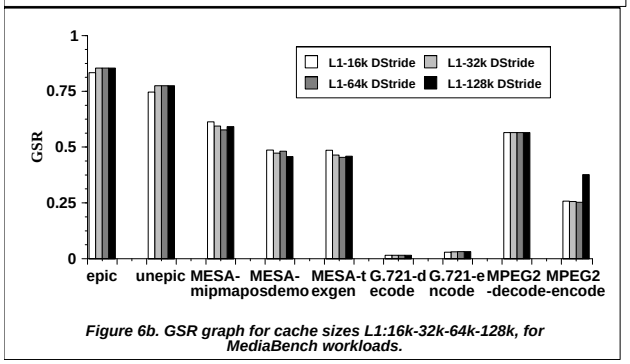

Figure 6b. GSR graph for cache sizes L1:16k-32k-64k-128k, for MediaBench workloads.

4.2.3. Effect of cache associativity on prefetching. Figure 7a shows that the RMCPI is not decreased once associativity reaches 2. Similarly the GSR also remains almost constant for all MediaBench workloads, as shown in Figure 7b. The reason for this are the same as those that explains the trends with varying cache sizes.

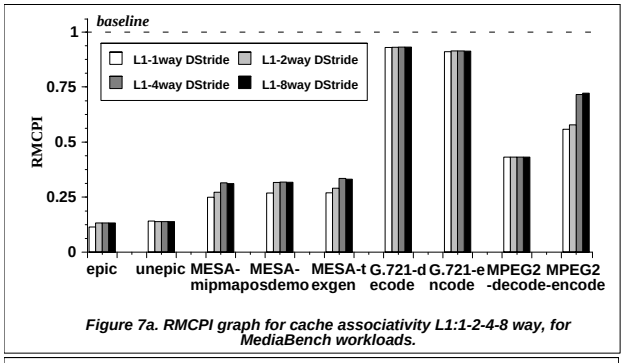

Figure 7a. RMCPI graph for cache associativity L1:1-2-4-8 way, for MediaBench workloads.

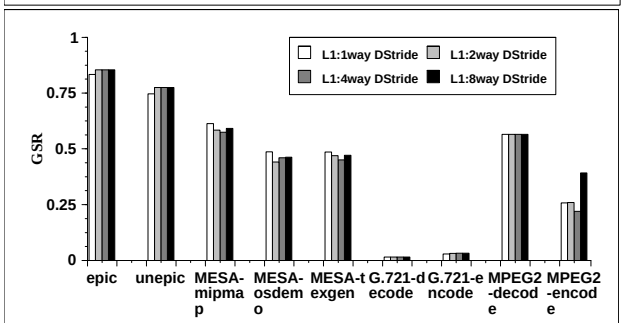

Figure 7b. GSR graph for cache associativity L1:1-2-4-8 way, for MediaBench workloads.

5. Conclusion

We have evaluated the proposed DStride prefetching model by multiple-issue execution driven simulation using SimpleScalar toolset primarily on MediaBench workloads. On the average, the overall relative memory penalty has been reduced by about 60-75% relative to the baseline system (without prefetching). The accuracy of the DStride predictor is about 50-75%, and there could be opportunities to improve it further by tuning the prefetcher parameters. Immediate future work will consist of carrying out a VLSI realization in order to asses cost and performance (in terms of operational times). Further study will also include adapting the design of the prefetcher to multi-threaded processors.

Acknowledgments

We would like to thank Sun Microsystems Singapore and Ed Smith at Sun Microsystems for the support provided in this work.

References

[1] Jean-Loup Baer, Tien-Fu Chen, "An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty," ACM, pp 176-186, 1991.

[2] Doug Joseph, Dirk Grunwald, "Prefetching Using Markov Predictors," IEEE Transaction on computers, Vol 48, No 2, Feb 1999.

[3] D.C Burger, T.M. Austin, "The SimpleScalar Tool Set, Version 2.0," Technical Report CS-TR-97-1342, Univ. of Wisconsin-Madison, Jun 1997.

[4] John Tse, Alan Jay Smith, "CPU Cache Prefetching: Timing Evaluation of Hardware Implementations," IEEE Transactions on Computers, Vol. 47, No 5, May 1998.

[5] T.Ozawa et al., "Cache Miss heuristics and Preloading techniques for General-Purpose Programs," Proc. 28th Ann. Int'l Symp. Microarchitecture, pp. 243-248, Nov. 1995.

[6] S.S.Pinter and A.Yoaz, "Tango: a hardware-based data prefetching technique for superscalar processors", Proc. 29th Ann. Int'l Symp. Microarchitecture, December 1996.

[7] www.mpeg.org

[8] John W.C. Fu, Janak H. Patel and Bob L. Janssens, "Stride Directed Prefetching in Scalar Processors", IEEE 1992.

[9] Fredrick Dahlgren and Per Stenstorm, "Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors", IEEE Trans. on Parallel and Distributed Systems, 1996.

[10] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers," Proc. 17th Int'l Symp. Computer Architecture, May 1990.

[11] Subbarao Palacharla and R.E. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement,", Proc. 21st Ann. Int'l Symp. Computer Architecture, pp. 211-222 Apr. 1994.

[12] Doug Joseph and Dirk Gurnwald, "Prefetching using Markov Predictors," Trans. on computers, Vol 48, No 2, Feb 1999.

[13] D. F. Zucker, M.J. Flynn and R.B Lee, "A Comparison of Hardware prefetching Techniques for MultiMedia Benchmarks," TR CSL-TR-95-683, Dec. 1995.

[14] Chunho Lee, M Potkonjak and W. H. Magione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," 1997.

[15] M.J. Charney and A.P Reeves, " Generalized Correlation Based Hardware Prefetching, " TR EE-CEG-95-1, Cornell Univ. Feb. 1995.

[16] H.Oehring, U.Sigmund, T.Ungerer, "MPEG2-Video Decompression on Simultaneous Multithreaded Multimedia Processors," IEEE, 1999.