In Proc, of the IEEE Int. Performance, Computing and Communications Conference, Phoenix, USA, Feb. 2000,

Hardware versus Hybrid Data Prefetching in Multimedia Processors: A Case Study

Andy D. Pimentel Louis O. Hertzberger Pieter Struik Pieter van der Wolf

Dept. of Computer Science
University of Amsterdam, The Netherlands

Philips Research Laboratories
The Netherlands

Abstract

Data prefetching is a promising technique for hiding the penalties due to compulsory cache misses. In this paper, we present a case study on two types of data prefetching in the context of multimedia processing: a purely hardware-based technique and a more low-cost hybrid hardware/software technique. Moreover, we also propose a technique for increasing the so-called prefetch distance in hardware prefetching and a scheme to reduce trashing in the data cache. Our results demonstrate that the low-cost hybrid prefetching scheme slightly outperforms hardware-based prefetching for the code segments for which both solutions have been applied, while hardware prefetching potentially allows more code to benefit from the prefetching.

1 Introduction

The memory design community has not been able to keep up with the rapid advances in microprocessor technology of the last two decades. As a consequence, a large gap between the performance of microprocessors and main memory developed. To a certain extent, caches are capable of reducing this performance gap. However, in some application domains, such as the increasingly popular class of multimedia applications, the benefits of caches are limited. In multimedia applications, calculations are often applied to *streams of data* (e.g. video streams). This means that the data is accessed in a regular fashion, i.e. using a fixed distance between two accesses. This distance is called the *stride*. Because stream processing usually exhibits no to little re-use of data, it suffers from a large number of compulsory data cache misses (caused by references that have not been referenced before).

Data prefetching is a technique that can be used to hide the latencies due to compulsory cache misses. By giving a *hint* to the memory system to try to bring data into the data cache before it is actually referenced, later binding memory references may complete much faster. Ideally, all these prefetched memory references will hit in the data cache, implying that prefetching should always retrieve the correct data and that this data is always available in time. But, even in the case the data is still being prefetched at the time it is referenced, there usually still is a performance gain with respect to a plain cache miss (the prefetched data is already on its way from main memory). However, the danger of prefetching is *trashing*: prefetched data may either be fetched too soon or it is not going to be used in the future. In both cases, the prefetched data may trash the data cache by replacing valuable data.

This paper presents a case study in which we evaluate two techniques for prefetching data streams: a technique which uses a pure hardware solution and a recently proposed hybrid technique which applies both hardware and software for prefetching. We have centered the case study around a typical and well-understood video-processing application on which a range of experiments are performed. The motivation for this work is to find cost-effective methods for reducing the average memory latency in multimedia processors, and in particular those focusing on the embedded market. The next section presents an overview of several popular data stream prefetching techniques. Section 3 describes the simulation model we have used in this study. This section also proposes two new features: an optimisation for hardware-based prefetching to increase the so-called prefetch distance and a technique to reduce cache trashing. In Section 4, we present experimental results of both hardware-based and hybrid prefetching when applied to our video-processing application. Section 5 concludes the paper.

2 Data stream prefetching

In this study, we focus on techniques that exploit the regularity of accesses in stream processing to prefetch the streams into the data cache. Because a wide variety of these so-called *stream prefetching* methods exists, we briefly discuss the different efforts that have been made in this field. To classify these efforts, we first identify the two actions of which stream prefetching is composed:

- *Stream detection*:
  Detect when an application is performing operations on data streams.

- *Synchronisation*:
  While a data stream is processed, prefetch requests for stream elements should be issued at proper moments in time. These prefetch requests should be controlled so that no or little trashing occurs.

Both detection and synchronisation can either be performed statically (by the compiler or programmer) or dynamically (in hardware), resulting in four combinations of which three are realizable (dynamic detection, static synchronisation is not possible). For the sake of convenience, we only refer to the compiler when describing static techniques.

Static detection, static synchronisation.

A straightforward way to prefetch data is to add a *scalar prefetch* instruction to the processor's instruction set [6]. This instruction, which is inserted in the code at strategic places by the compiler, instructs the data cache to prefetch a certain cache block. So, the compiler must detect where prefetch instructions have to be inserted such that the prefetched data arrives in time and, additionally, it should insert enough prefetch instructions to prefetch the entire data stream.

The major disadvantage of this *software prefetching* is the increase of the number of instructions. For each cache block that has to be prefetched, at least one prefetch instruction has to be executed. Moreover, the prefetch instructions may also affect compiler optimisations, such as loop unrolling [4]. Figure 1(a) shows an example of software prefetching. In this paper, we will not consider this type of prefetching.

Dynamic detection, dynamic synchronisation.

In *pure hardware prefetching*, both the detection of the streams and the synchronisation of the prefetch requests are performed at run-time. A well-known hardware prefetching method, proposed by Fu and Patel [3], introduces a hardware table which records the history of memory references to identify streams and to predict future references. This table, called the Stride Prediction Table (SPT), stores the instruction address (i.e. the program counter value) of memory references together with the data address that is referenced. At a new memory reference, the actual value of the program counter (PC) is searched for in the SPT. If it is found, then the stride can be calculated by subtracting the data address stored in the SPT-entry from the data address of the current reference. Subsequently, a request is issued to prefetch data from the location which is anticipated to be accessed next, being the current data address plus the stride. So, this method synchronises the issuing of the prefetches using the PC.

Chen and Baer proposed several optimisations to this scheme [2]. In order to reduce the number of erroneous

```
prefetch(&a[0]);
for (i=0; i<N-1; i++)
{
    prefetch(&a[i+1]);
    sum = a[i] + sum;
}
sum = a[i] + sum;
```

(**a**)

```
streamprefetch(&a[0], N, 4, 1);
for (i=0; i<N; i++)
    sum = a[i] + sum;
```

(**b**)

Figure 1. Software prefetching (a) and hybrid hardware/software prefetching (b). The parameters of the hybrid stream prefetch instruction are respectively: the start address, the number of elements to prefetch, the stride and the preferred runahead.

prefetches, they added state information to each SPT entry. This state indicates whether or not a prefetch request should be issued at an SPT hit. For example, the state of an SPT-entry is valid (implying it can issue prefetch requests) only when a constant stride is measured within the stream. By doing so, irregular access patterns do not cause erroneous prefetches as the SPT just ignores these accesses. Chen and Baer also describe a technique to use a so-called lookahead program counter (LA-PC) instead of the normal PC to search the SPT. With the help of the processor's branch prediction table, the LA-PC runs ahead with respect to the PC and therefore increases the *prefetch distance*. The prefetch distance, or *runahead*, is the distance the prefetches run ahead with respect to the actual references within the stream. A larger runahead gives the memory system more time to prefetch the data. In this paper, we use the term runahead to indicate how many stream references the prefetching runs ahead.

There are several consequences of the fact that SPT-based prefetching relies on the PC for the identification of streams and the synchronisation of prefetches. First, the SPT should be reasonably large as every memory operation may indicate the start of a stream. Second, the PC needs to be routed to the SPT logic in the data cache, thereby affecting the processor's logic. Third, compiler optimisations like loop unrolling may affect the performance of SPT-based prefetching. Loop unrolling splits one stream into a number of sub-streams which all have to fit in the SPT. In addition, SPT-based prefetching suffers from two more drawbacks: there is a delay before a stream is actually detected (it takes at least two memory references to determine the stride) and the SPT may issue erroneous prefetch requests (e.g. it can issue a request for the cache block beyond the end of a stream).

Static detection, dynamic synchronisation.

Prefetching can also be done using a hybrid hardware and software solution such that the best of both worlds are combined. For example, the problem of the large number of extra instructions as experienced in pure software prefetching is solved by synchronising the prefetch requests dynamically rather than statically. Moreover, by statically detecting

streams, there is more control over the prefetching than in case of dynamic detection. As a consequence, the number of erroneously prefetched cache blocks can be reduced to zero. Additionally, the amount of required hardware resources for hybrid prefetching is smaller than for SPT-based prefetching. This is because the latter form of prefetching also records information on memory references that do not behave as a stream. Thus, hybrid prefetching generally is more low-cost than SPT-based prefetching, which is important when targeting, for example, the embedded market.

In hybrid prefetching, the compiler detects streams within a program and inserts special *stream prefetch* instructions at the appropriate places. The stream prefetch instructions trigger a special piece of hardware to issue a prefetch request from time to time. Like in SPT-based prefetching, the synchronisation of prefetch requests (i.e. determining *when* a request needs to be issued) is usually performed using the program counter [1]. In this *PC-synchronised hybrid prefetching*, the stream prefetch instruction initialises an entry in a special hardware table, called the Prefetch Information Table (PIT). The instruction provides the PIT with the synchronising instruction address which should trigger a prefetch request, the data address at which the prefetching should start, the stride with which should be prefetched and a count specifying the number of prefetches that should be performed. In addition, the preferred runahead can also be specified. At each memory access, it is checked if the program counter (PC) is present in the PIT. If the PC is found in the PIT, then a prefetch request is issued and the relevant information (e.g. the prefetch address and count) in the PIT entry is updated. Figure 1(b) shows an example of hybrid prefetching.

This prefetch technique still shares several drawbacks with pure hardware prefetching. First, the PC must still be routed to the PIT logic. Second, PC-synchronised hybrid prefetching is also affected by loop unrolling, creating a number of smaller sub-streams which all should fit in the PIT. To overcome these drawbacks, we have proposed a new hybrid prefetching scheme which is based on *data address synchronisation* rather than on PC-based synchronisation [8]. Data-synchronised prefetching uses a PIT which is almost identical to that from PC-synchronised prefetching. But rather than specifying an instruction address that synchronises the prefetches, the starting data address of the stream is specified for the synchronisation. Thus, instead of matching the PC, the data address of memory references is used to determine if there is a PIT hit. At each hit, a prefetch request is issued after which the synchronisation and prefetch addresses of the PIT entry are updated using the stride.

Compared to PC-synchronised prefetching, our data address synchronisation scheme has three important advantages. First, the PC does not need to be forwarded to the PIT. Second, data address synchronisation may lead to a significantly smaller PIT because this technique is not affected by compiler optimisations. Unrolling a loop with a stream, for instance, does not result in a number of smaller sub-streams in the PIT; there is still one PIT entry for the unrolled stream. Finally, data-synchronised prefetching is more robust than its PC-synchronised counterpart. More specifically, data address synchronisation determines *where* the processor is accessing a stream, rather than determining *that* the processor is accessing a stream like PC-based synchronisation does. This allows, for example, prefetching with a stride that is different from the one used for the actual referencing of the stream [5]. However, there is one drawback of data-synchronised prefetching. The synchronisation data addresses which are searched for in the PIT change dynamically over time. This implies that the PIT must be implemented using a fully associative memory. By contrast, a PIT in PC-synchronised prefetching can be implemented as a set-associative table. But, since the PIT can be kept small with data-synchronised prefetching, its fully associative implementation should not be a problem.

3 The simulation model

In this study, we used a trace-driven simulation model of the TriMedia VLIW multimedia processor architecture [7, 9]. The VLIW instructions contain five slots in which RISC-like operations can be scheduled. At most two of the operation slots can be used for memory accesses (load or store operations). In our simulator, instructions take by default 1 cycle to execute (the hitratio of the instruction cache is assumed to be perfect). Memory references are, however, explicitly simulated by a data cache and bus model which account for the latencies in a cycle-accurate way.

The data cache model models a non-blocking 16 Kbyte 8-way set-associative cache with 64-byte cache blocks and a Hierarchical-LRU (HLRU) replacement strategy. By default, the cache applies the write-back and fetch-on-write policies for write operations. But it also provides a *write-validate* policy, which allocates a cache block on a write-miss but does not fetch the data. For this purpose, a valid bit is kept for each separate byte within a cache block. We have included the write-validate policy because stream-processing applications often write their results in the form of an output stream. In that case, it is unnecessary to fetch a write-missed cache block as it will be overwritten anyway.

To allow prefetching, the data cache model uses a Prefetch Queue (PQ) in which issued prefetch requests wait to be handled. The PQ is a FIFO buffer which ignores new prefetch requests when it is full. The data cache model includes a 4-way set-associative SPT for hardware prefetching and an 8-way fully associative PIT for hybrid prefetching. Both tables apply LRU replacement. The SPT uses state bits, similar to those from Chen and Baer [2], to guarantee that prefetch requests are issued only if a stream exhibits a constant stride.

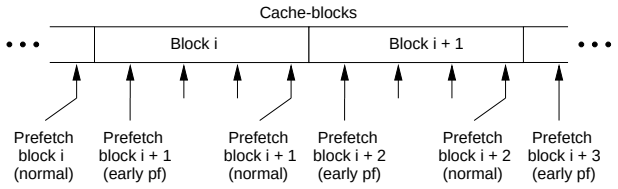

Figure 2. The difference between normal SPT-based prefetching and prefetching using the early-prefetch mode.

3.1 Two new techniques

In our model, we have also applied two new techniques [5]. The first technique addresses the potential weakness of the traditional SPT-based prefetching method associated with the timing of prefetches, that is, the prefetch requests are issued only one iteration before the data is really needed. If the loop body is too small, the prefetched data may arrive too late for the next access. Unlike Chen and Baer, who introduce a lookahead-PC [2] to solve this problem, we have chosen for a simpler approach. Our technique, which is called *early-prefetch* [5], exploits the situations in which the stride is smaller than the cache block size. Whereas normal SPT-based prefetching ignores a prefetch request that still refers to the same cache block as the previous request, early-prefetch simply issues a prefetch request for the succeeding cache block in that case. In Figure 2, the early-prefetch concept is illustrated. The arrows indicate the stream memory-references made by the program. The figure shows that normal SPT-based prefetching issues the prefetch requests one iteration before a next cache block is accessed (indicated by the arrows tagged with "normal") while the early-prefetch scheme issues a prefetch request for the succeeding block when the first element in a block is accessed. For small strides, the early-prefetch technique may increase the prefetch distance (i.e. runahead) significantly. On the other hand, early-prefetching will often erroneously prefetch one extra cache block behind the end of the stream. Moreover, early-prefetching is only beneficial when the streams are contiguously laid out over the cache blocks. If the stride within a stream is equal to or larger than the cache block size, then early-prefetching has no effect.

The other new technique we included in our model tries to reduce the effect of trashing (as the prefetched data may remove valuable, persistent data from the cache). To this end, the cache model allows to specify a sub-set of cache blocks which may be the target for prefetched data. One could, for instance, configure only two of the 8 blocks in a cache set for prefetching. In that case, the remaining six blocks within the set are strictly used for normal data references and cannot be trashed by prefetches. Normal data references can, however, use the prefetch blocks as well. For the prefetch blocks, an additional replacement strategy is used, which is LRU. Prefetches update both the global strategy (i.e. HLRU) and their "local" prefetch strategy. As a consequence, normal data references will try to avoid prefetch blocks when a lot of prefetching is done, thereby almost guaranteeing exclusive access to the prefetch blocks by prefetches only. On the other hand, if there is not much prefetch activity, the cache is free to use all the blocks within a set for normal data references.

4 Experiments

To evaluate hardware-based and hybrid prefetching, we have performed a simulation study using a typical and well-understood video-processing application, called Median. The kernel of this program processes the odd and even frames of an interlaced video stream in order to produce non-interlaced frames. For all our experiments, we applied Median to a 200×140 image. Because we focus on a single benchmark, the purpose of this study is to gain insight into the *relative* behaviour of the two prefetching methods. By no means, we try to predict absolute performance improvements by extrapolating the results from the Median benchmark.

Table 1 shows the data cache missratio and the Data Cache Penalty (*DCP*) for Median without prefetching. The DCP value refers to the average number of cycles per memory reference the processor stalls for the data cache. The table gives the results for both the fetch-on-write and write-validate policies. The results from Table 1 indicate that Median benefits from a write-validate data cache. As the missratios obtained by the fetch-on-write policy are only marginally lower than the write-validate missratios, this suggests that there rarely are read accesses to cache blocks into which some other item has been written. So, by not fetching cache blocks at write misses (i.e. write-validate) valuable memory bandwidth is saved. Consequently, the DCP for the write-validate policy is roughly 15% lower than that for a fetch-on-write cache. Our goal is to reduce the DCP even further using prefetching.

For hybrid prefetching, we instrumented the kernel of Median with stream prefetch instructions. Hence, prefetching only occurs during the execution of this kernel. By contrast, with hardware prefetching, prefetch requests can be issued during the execution of the *whole* program. As the Median benchmark also includes several input and output routines for the video streams, these code segments may benefit from hardware prefetching as well. On the other hand, prefetching

Write-validate		Fetch-on-write	
Dcache$_{missratio}$	DCP	Dcache$_{missratio}$	DCP
1.68%	0.157	1.66%	0.185

Table 1. Missratio and DCP for a non-prefetching data cache with fetch-on-write and write-validate.

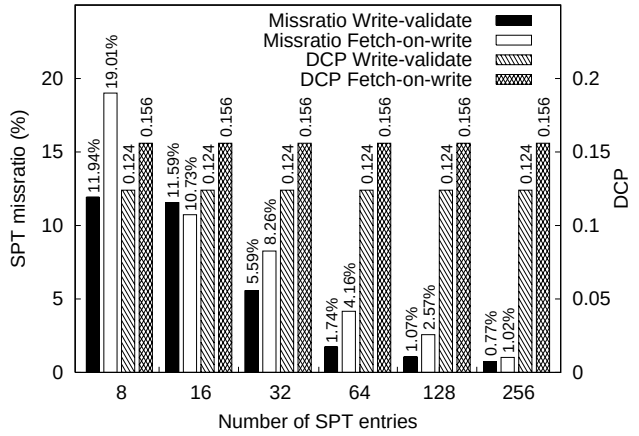

Figure 3. SPT missratio and DCP for a fetch-on-write and write-validate cache.

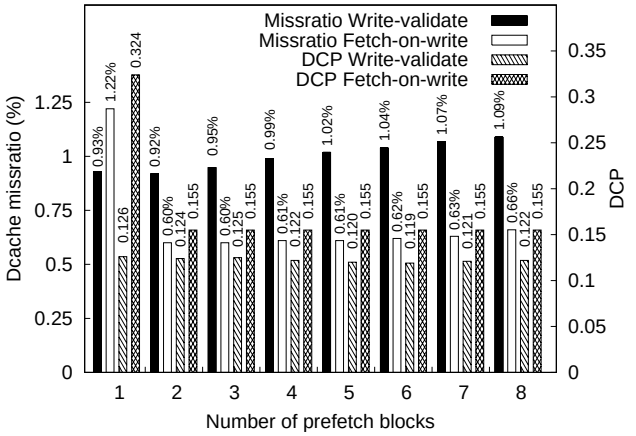

Figure 4. The performance when varying the number of prefetch cache blocks for fetch-on-write and write-validate.

during the whole program may also increase the amount of trashing within the data cache. Which of these two effects prevails, is entirely dependent on the type of application.

4.1 Hardware prefetching

For hardware prefetching, a 4-way set-associative SPT is used which may range in size from 8 to 256 entries. As the simulated VLIW instructions can hold two memory references, which may interfere with each other in the SPT, we have experimented with several implementations of the SPT [5]. We found that a single-banked SPT, which is indexed by the formula SPT[(PC + instr_slot) *mod* SPTSIZE], yields the highest SPT hitrates. Here, the *instr_slot* refers to the memory-operation slot within the VLIW instruction and equals to 0 or 1. The addition of *instr_slot* prevents aliasing between two memory operations within one VLIW instruction. Throughout the remainder of this paper, we assume that this particular SPT implementation is used. By default, it operates without early-prefetch. Furthermore, unless stated otherwise, the data cache features a single-entry Prefetch Queue (PQ) and contains two prefetch blocks per cache set in order to reduce trashing (see Section 3.1).

In Figure 3, the SPT missratios are shown for both the fetch-on-write and write-validate policies. The SPT missratio is the percentage of "prefetchable" memory references that miss in the SPT. In fetch-on-write, both loads and stores are prefetchable whereas in write-validate only loads are prefetchable. As the SPT missratio is no indicator of real performance, the figure also shows the obtained DCP values for the same experiment. The results clearly illustrate that prefetching has significantly decreased the DCP values as compared to the values listed in Table 1. Not surprisingly, the best improvements are for the write-validate policy. In the fetch-on-write policy, a lot of prefetches are, when compared to the write-validate policy, not effective as they are used for write operations only and not for read accesses. These prefetches did, however, prevent other (effective) prefetches from being executed, thereby limiting the DCP reduction.

Furthermore, Figure 3 also indicates that the SPT missratios of write-validate generally are somewhat lower than the ones of fetch-on-write. This is probably due to several write operations which are not part of a stream but still trash the SPT. Another observation is that reducing the SPT missratio (by enlarging the SPT) does not result in a lower DCP. This can be explained by the fact that, for Median, the number of prefetches issued by both small and large SPTs is almost identical (these numbers are not shown). So, the lower missratio of large SPTs does not result in extra effective prefetches.

Figure 4 shows the cache missratio and the DCP when the number of prefetch blocks per set in the cache are varied. Limiting the number of prefetch blocks can reduce the amount of cache trashing due to prefetching. In this experiment, a 128-entry SPT is used. The results show that one prefetch block is not sufficient when applying the fetch-on-write policy. For the other configurations, the performance differences are marginal and there cannot be detected a clear trend. This suggests that Median does not use a lot of other, more persistent, data besides its video streams. So, to evaluate our proposed trashing reduction feature, a more quantitative study with other benchmarks is needed.

For a write-validate cache and a 128-entry SPT, Table 2 shows the results when varying the Prefetch Queue (PQ) size. It shows both the number of cancelled prefetch requests due to a full PQ and the performance impact. Table 2 suggests that increasing the PQ size, which results in less cancellations, only marginally improves the performance. This is caused by the small runahead with which the SPT operates:

PQ size	# Cancellations	Dcache$_{missratio}$	DCP
1	4044	0.92%	0.124
2	3140	0.92%	0.123
4	1143	0.91%	0.123
8	8	0.87%	0.121

Table 2. Performance impact of a larger PQ when using write-validate and a 128-entry SPT.

although the larger PQ size avoids a large number of the cancellations, most of the "saved" prefetch requests are handled too late. This means that the required cache blocks have already been referenced at the time the prefetch requests are handled, after which the requests are discarded after all.

So far, the prefetch results are obtained by a model in which prefetching is turned on during the execution of the whole program. To allow a comparison with hybrid prefetching, which only prefetches during the execution of Median's kernel, we also measured the impact when hardware prefetching is enabled only in the kernel. Figure 5 shows the results of this experiment for a 128-entry SPT, a write-validate cache with 2 prefetch blocks per set and an 8-entry PQ. The leftmost set of bars gives the results without prefetching, whereas the middle set of bars shows the results for both prefetching during the whole program and kernel-only prefetching. Note that the missratio and DCP are *overall* values, i.e. measured for the whole program. With kernel-only prefetching, the missratio and DCP slightly improve (compared to no-prefetching) but not to the level when prefetching is enabled during the whole program. More precisely, we measured a DCP improvement of 23% when prefetching during the whole program (0.157 to 0.121), while the DCP only improves with 9% when strictly prefetching in the kernel (0.157 to 0.143). So, most of the performance gain due to SPT-based prefetching is obtained during the execution of auxiliary functions rather than in the kernel.

Figure 5 also shows the results for the early-prefetch optimisation (rightmost set of bars), indicating that this optimisation is highly effective for Median. It decreases the DCP for kernel prefetching with 27% compared to normal prefetching. Comparing this to execution without prefetching, this resolves into a DCP reduction of 34%. If prefetching is allowed during the whole program, then the improvements are even more substantial. In that case, the DCP has decreased with 65% compared to normal prefetching and 73% compared to the no-prefetch results. These large improvements indicate that with normal prefetching a lot of prefetches are started (or finished) too late due to a poor runahead. This corresponds to our findings from Table 2.

4.2 Hybrid prefetching

For the hybrid prefetching experiments, we again use a write-validate cache model with two prefetch blocks per set.

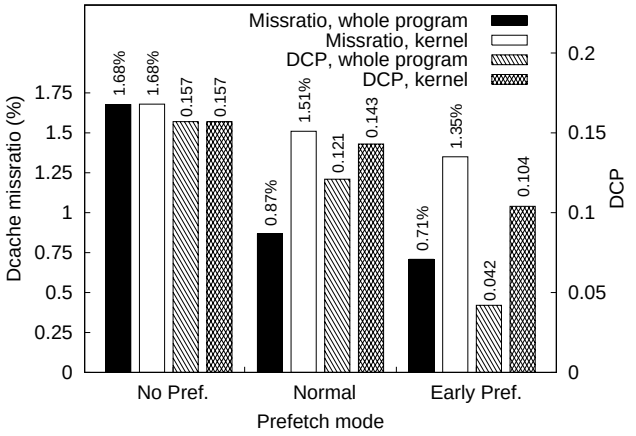

Figure 5. Performance of overall prefetching and kernel-only prefetching for write-validate, a 128-entry SPT and an 8-entry PQ. The results of the early-prefetch optimisation are also shown.

By default, we have parameterised the stream prefetch instructions with an arbitrary runahead of 3, being the number of stream references the prefetching is ahead of the actual stream processing. In this paper, we only present the data-synchronised hybrid prefetching results. In [5], we demonstrated that the performance potential of PC-synchronised and data-synchronised prefetching is identical while the latter offers more advantages implementation-wise. Note that the following experiments should be compared with the kernel-only hardware prefetching results from the previous section.

Figure 6 shows the results of (data-synchronised) hybrid prefetching. The leftmost set of bars gives the performance of a non-prefetching data cache, while the remaining bars present the results for a prefetching cache with different PQ sizes. The "Stream missratio" gives the missratio measured for the data stream elements only. From Figure 6 can be seen that the PQ size only marginally affects the performance. Only a single-entry PQ seems to be too small.

When comparing these results to the ones in Figure 5, one can conclude that hybrid prefetching with a runahead of 3 yields lower DCP values than normal, kernel-only hardware prefetching. More specifically, hybrid prefetching improves the DCP with about 16% compared to (normal) kernel-only hardware prefetching. However, hardware prefetching using the early-prefetch optimisation still outperforms hybrid prefetching with a runahead of 3.

In Figure 7, the effect of varying the runahead is shown. Again, the leftmost bars give the results for a non-prefetching cache. The results indicate that the performance steadily improves when increasing the runahead until a runahead of 12 is reached. The DCP obtained with a runahead of 12 is just slightly lower than the one obtained by early-prefetch hardware prefetching (see Figure 5). Also, the Stream mis-

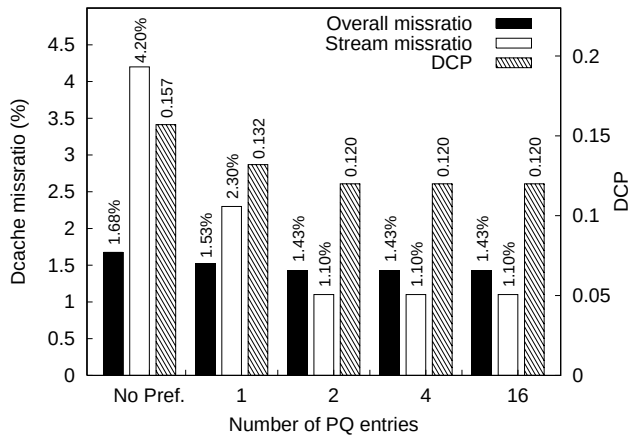

Figure 6. Performance of data-synchronised hybrid prefetching using a runahead of 3 and write-validate.

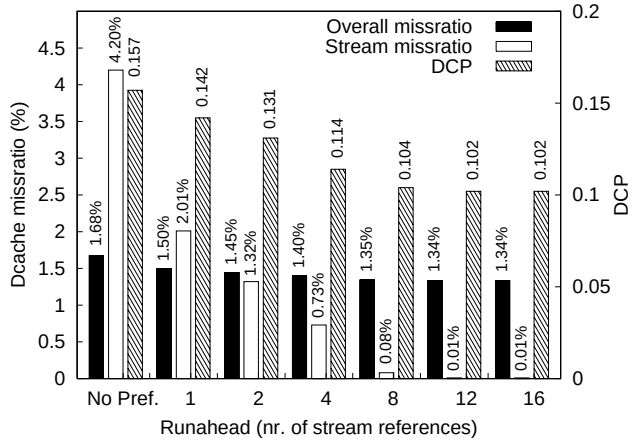

Figure 7. Performance of data-synchronised hybrid prefetching using a 16-entry PQ and write-validate.

sratio indicates that for large runaheads, there is almost no cache miss on the stream elements. It is not surprising that both hybrid prefetching with a runahead of 12 and early-prefetch hardware prefetching perform similarly: both methods more or less guarantee that the succeeding cache block is prefetched as soon as its predecessor is referenced. For example, a runahead of 12 means a prefetch distance of $12 * 8 = 96$ bytes (the stride in Median equals to 8 bytes), which is larger than the 64-byte blocksize of the cache. So, from Figure 7 can be concluded that a reasonably large runahead is essential for effective prefetching in the Median benchmark.

So far, we did not mention the effect of prefetching on the bus utilisation. For Median, the investigated prefetching techniques are nearly perfect, implying that they issue a small number of erroneous prefetch requests. As a result, we measured only tiny differences ($< 1\%$) between the bus utilisation of non-prefetching and prefetching data caches.

5 Conclusions

In this paper, we studied pure hardware-based and hybrid hardware/software stream prefetching for a video processing application. The results indicate that prefetching is an effective latency hiding technique for the studied workload. We measured reductions of the average data cache penalty per memory reference of up to 73%. A second observation is that pure hardware prefetching, with our proposed "early-prefetch" optimisation, obtains the largest improvements for the applied workload. This is because hardware prefetching allows for prefetching during the execution of the whole application. By contrast, hybrid prefetching only allows for prefetching during code segments which were explicitly instrumented to do so.

When focusing on the code segments in which prefetching is enabled for both prefetch techniques, we found the hybrid prefetching technique to be slightly more effective than hardware prefetching (although the differences are minimal). This is despite the fact that hybrid prefetching requires less and simpler hardware than pure hardware prefetching and is therefore cheaper to implement.

Future work should extend this study with a representative set of multimedia benchmarks in order to perform a more quantitative analysis of the absolute prefetch performance.

References

[1] T.-F. Chen. An effective programmable prefetch engine for on-chip caches. In *Proc. of the 28th Int. Symposium on Microarchitecture*, pages 237–242, Nov. 1995.

[2] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[3] J. W. C. Fu and J. H. Patel. Stride directed prefetching in scalar processors. In *Proc. of the 25th Int. Symposium on Microarchitecture*, pages 102–110, 1992.

[4] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Oct. 1992.

[5] A. D. Pimentel. *A Computer Architecture Workbench*. PhD thesis, Dept. of Comp. Science, Univ. of Amsterdam, Dec. 1998.

[6] V. Santhanam, E. H. Gornish, and W. Hsu. Data prefetching on the HP PA-8000. In *24th Int. Symposium on Computer Architecture*, pages 264–273, June 1997.

[7] G. A. Slavenburg, S. Rathnam, and H. Dijkstra. The TriMedia TM-1 PCI VLIW media processor. In *Proc. of Hot Chips 8*, pages 171–177, Aug. 1996.

[8] P. Struik, P. van der Wolf, and A. D. Pimentel. A combined hardware/software solution for stream prefetching in multimedia applications. In *Proc. of SPIE Multimedia Hardware Architectures*, pages 120–130, Jan. 1998.

[9] J. T. J. van Eijndhoven, F. W. Sijstermans, K. A. Vissers, E. J. D. Pol, M. J. A. Tromp, P. Struik, R. H. J. Bloks, P. van der Wolf, A. D. Pimentel, and H. P. E. Vranken. TriMedia CPU64 architecture. In *Proc. of the IEEE ICCD '99*, Oct. 1999.