A Combined Hardware/Software Solution for Stream Prefetching in Multimedia Applications

Pieter Struik[a], Pieter van der Wolf[a], and Andy D. Pimentel[b]

[a]Philips Research Laboratories, Eindhoven, the Netherlands
[b]University of Amsterdam, the Netherlands

ABSTRACT

Prefetch techniques may, in general, be applied to reduce the miss rate of a processor's data cache and thereby improve the overall performance of the processor. More in particular, stream prefetch techniques can be applied to prefetch data streams that are often encountered in multimedia applications. Stream prefetch techniques exploit the fact that data from such streams are often accessed in a regular fashion. Implementing a stream prefetch technique involves two issues, viz. stream detection and stream prefetching. Both problems can be solved both in hardware and in software. This paper presents a combined hardware/software stream prefetch technique. A special stream-prefetch instruction is introduced to alert the hardware that load instructions access a data stream. Subsequently, prefetching is handled by the hardware automatically in such a way that the rate at which the data is prefetched is synchronized with the rate at which the prefetched data is processed by the application. These kinds of stream prefetch techniques have been proposed earlier but use instruction addresses for synchronization. The technique that is introduced in this paper uses a different synchronization mechanism that does not suffer from drawbacks of instruction address synchronization.

Keywords: prefetching, stream prefetching, cache, hardware/software solution, synchronization, compiler optimizations

1. INTRODUCTION

As technology advances the increase in memory speed lags the increase in processor speed. Therefore, cache misses result in increasing miss penalties that may decrease processor performance significantly. Scaling the cache size does in general improve cache performance, but for relatively large caches the law of diminishing return will play a role and there may be more effective options to utilize scarce silicon resources. More specifically, multimedia applications that process data streams suffer from compulsory misses. Stream processing has limited reuse of data and therefore larger caches do not help since they will still suffer from these compulsory misses. Prefetching is a technique that in general improves processor performance by hiding the large memory latency of cache misses. A prefetching cache loads data in advance from main memory or a L2 cache while the central processing unit still executes other instructions. Thus, prefetching exploits parallelism.

Many prefetch techniques have already been proposed and applied. We discuss a stream prefetch technique that can be applied to prefetch data streams that are often encountered in multimedia applications. This stream prefetch technique is a combined hardware/software technique since it uses an instruction (software) to signal a hardware prefetch engine to automatically prefetch a stream of data elements. This is done in such a way that the rate at which the data is prefetched is synchronized with the rate at which the data is processed. Stream prefetching techniques that synchronize on instruction addresses are known. This paper discusses a stream prefetching technique that synchronizes on the data addresses of the load instructions. This technique does not introduce overhead to the critical path and is expected to have a cost-effective implementation.

Correspondence: P.Struik. Other author information: P.S.; e-mail: struik@natlab.research.philips.com; address: prof. Holstlaan 4, WL11, 5656 AA Eindhoven, the Netherlands. P.v.d.W.; e-mail: vdwolf@natlab.research.philips.com. A.D.P.; e-mail: andy@wins.uva.nl.

This paper is organized as follows. Section 2 introduces prefetching and lists a number of requirements that we impose on a good prefetch technique. Section 3 gives an on overview of existing prefetch techniques and existing stream prefetch techniques, in particular. In addition, it introduces our combined hardware/software technique that synchronizes on data addresses of load instructions. Section 4 motivates why synchronization on data addresses is used instead of an existing technique that synchronizes on instruction addresses. The basic hardware architecture that implements stream prefetching is discussed in Section 5. Next, Section 6 presents experimental results. Finally, Section 7 gives some concluding remarks.

2. PROBLEM STATEMENT

Prefetch techniques are applied to hide the large memory latencies of cache misses. Prefetching improves the average memory access time by reducing the miss rate. This is achieved by loading data into the cache before it is referenced for the first time. Thus cache misses are turned into cache hits.

A first requirement for a good prefetch technique is that it has to be effective. The effectiveness of a prefetch technique depends on how well it is able to eliminate cache misses. At first, prefetched data must be stored in the cache in time such that the reference to it results in a hit. Secondly, prefetching must avoid the side effect that it replaces a cache block[1] that needs to be accessed before the prefetched data is accessed. Otherwise, there will still be cache misses. Finally, the prefetch technique should not load data into the cache that is not needed at all (thrashing). To put it differently, a 'perfect prefetch' fetches data that is really needed; it does this in time and without interfering with other memory accesses.

A second requirement that we impose on a good prefetch technique is that it should have a cost-effective implementation. In this respect, cost-effectiveness means that the technique needs only a limited chip area for its realization. In addition, it means that applying the prefetch technique does not result in a performance degradation, for instance due to the instruction overhead that is introduced by the technique.

A third requirement for a good prefetch technique is that it should be easily applicable by the programmer. We prefer a technique that can be applied by a programmer at the source code level without taking into account all kinds of details of the implementation platform (like, e.g., the cache architecture or compiler optimizations).

In this paper, we focus our attention on prefetching of stream data in the context of multimedia applications, although scientific code can benefit from stream prefetch techniques as well. Examples of streams are audio/video streams or data streams that are encountered in MPEG applications. Processing of these kinds of streams typically involves a loop in which each iteration one element or a limited number of elements are processed. For a stream $S = \{s_0, s_1, s_2, \ldots\}$, the data addresses of adjacent elements s_i and s_{i+1} usually differ by a fixed stride d. This means that the data address of a stream element can be calculated from the data address of the stream's first element, say address A_0, and its index i in the stream: element s_i has data address $A_i = A_0 + i*d$. Stream prefetching exploits this regularity of the stream's data addresses.

With respect to the realization of stream prefetching we consider two important aspects:

1. *Detection.* Before being able to prefetch stream elements one has to determine which data streams are present in an application.

2. *Synchronization.* While processing a stream, prefetches (or prefetch requests) must be generated on at proper moments in time. In addition, the generation of prefetches needs to be done in a controlled way, such that no thrashing occurs because of data elements that are prefetched too early in time.

The next section gives an overview of existing prefetch techniques and introduces the prefetch technique that we present in this paper

[1] also called cache line

3. PREFETCH TECHNIQUES

Many prefetch techniques are known. Each technique differs in the way detection and synchronization are implemented. Both hardware (or run time) and software solutions are possible for implementing the detection of streams and the issuing of prefetch requests.

In a pure hardware solution, the hardware monitors the load/store operations that are handled by the cache and decides which accesses are parts of a stream. Once a stream has been identified, prefetches are generated. An example of such a technique is the stream buffer [1] where on a miss in a block with block address *k* subsequent blocks *k+1*, *k+2*, … are being fetched and placed into a separate FIFO queue that is called a stream buffer. Observe that in this simple scheme, there is no stream detection. However, when processing a stream with a small stride (to be more precise, a stride that is at most the block size) subsequent elements are in the same or in subsequent cache blocks. And thus, a stream buffer can be used for stream prefetching. In order to support prefetching of multiple streams, multiple FIFO queues for prefetched data can be introduced. This is called a multi-way stream buffer. A stream buffer also generates prefetches for non-stream data, which may result in unused blocks that are being fetched. Unnecessary prefetches can be avoided by introducing some kind of filtering technique [2]. Such a filtering technique is used to detect the regular access pattern of a stream. Prefetches are generated for streams only. Based on this technique a number of different hardware realizations are possible. Such an implementation usually involves a prediction table that records whether or not a specific memory operation operates on stream data or not. This prediction table is a large table since every load operation potentially is the start of a stream access. When three subsequent data addresses a_0, a_1, and a_2 satisfy $(a_1 - a_0) = (a_2 - a_1)$, a stream has been detected with stride $d = (a_1 - a_0)$ and a prefetch request for address $a_3 = a_2 + d$ can be generated. In the prediction table, memory operations (e.g. a load instruction) are identified by the instruction address of the operation. In this way, prefetching of multiple streams (at most the number of table entries) can be supported. The table can also be used for synchronization purposes. When each memory operation results in the issuing of at most one prefetch request, the number of pending prefetches is limited. We call this *synchronization on instruction address*. Such a scheme is applied in the stream cache [3], where a separate cache is introduced to store prefetched data blocks.

In a pure software solution, the programmer is responsible for inserting prefetch instructions in the code. Which data streams are prefetched is a design decision of the programmer. A simple implementation of stream prefetching is by prefetching a data item one iteration (or a few iterations) before it is processed, like in the code fragment below

```
for ( i=0; i!=N; i++ )
   {
     ...
     prefetch( &s[i + R] );
     process( s[i] );
     ...
   }
```

where R determines how much prefetching ‘runs ahead’. A disadvantage of this approach is that a prefetch instruction is executed for each individual stream element. In general, there are many stream elements in a single cache block. In particular, this holds for the common case of a unit stride. Without any hardware provisions for reducing the number of prefetches that are executed, a solution for this problem is to rewrite the code such that each cache block is only fetched once, but in time. The latter requires that one has to take the relative block position of the first stream element into account. In the context of prefetching multiple streams, code rewriting becomes even more cumbersome [4]. A more advanced software solution is a compile-profile-compile solution [5] where during application profiling a hardware stream detection technique is simulated to identify data streams. After that, during the second compilation stage, prefetch instructions are automatically added, without any necessity for source code rewriting.

In a combined hardware/software solution, stream detection is done in software whereas the generation of prefetches (and synchronization) is done in hardware. Like the pure hardware solution, a table (called *prefetch table* here) is introduced to keep track of the process of issuing the prefetch requests. Again, there is one table entry per stream. Whereas the pure hardware solution fills an entry in the prediction table each time a ‘new’ load instruction is executed, the hw/sw stream prefetch technique uses an instruction [6] to fill the prediction table. Such a *stream prefetch instruction* is inserted before the loop that processes the corresponding stream, like in the following code fragment

```
streamprefetch ( S, .......... );
for ( i=0; i!=N; i++ )
   {
     ...
     process( s[i] );
     ...
   }
```

Either the programmer inserts a stream prefetch instruction in the source code or a compile-profile-compile solution is used to generate this instruction automatically. A stream prefetch instruction has at least the following arguments: an identification of the stream, the stride, and the stream size. In addition, there may be other arguments, like e.g. the 'run ahead'. There are a number of ways to identify a stream. How this is done also determines how to synchronize the issuing of prefetch requests. One possibility is to identify a stream by the instruction address of the memory operation (often a load operation) that operates on it [6]. In that case, prefetching is controlled by *synchronization on the instruction address*. Each time the memory operation's instruction address hits in the prefetch table it is determined whether or not to issue a prefetch. The instruction address of a memory operation is one of the parameters of the streamprefetch instruction. Notice that the issue of determining the instruction address of an operation at a source code level is far from trivial. Another possibility of identifying a stream is by the data addresses of its elements. This implies that prefetching is controlled by *synchronization on data addresses*. At a source code level the data address of the first data element of a stream is readily expressed: in C it would be something like `&s[0]` (or simply `s`).

The main advantage of a combined hardware/software technique over pure hardware techniques is the fact that the latter method needs a significantly larger prediction table since it must also record information on memory accesses that *may* possibly behave as a stream. The prefetch table of the hw/sw technique only records those streams for which prefetching is considered to be effective. Our experiments have shown that a prefetch table that supports at most 16 streams is considered to be large enough whereas the pure hardware technique requires a table of at least 128 entries.

The advantages of a combined hardware/software technique over pure software techniques are that there is no instruction overhead (in the loop body) introduced by scalar (i.e. one element) prefetch instructions, the hardware handles cache block alignment issues that are difficult to handle at the source code level, and loop code optimizations that may be performed do not interfere with prefetching.

4. WHY SYNCHRONIZATION ON DATA ADDRESSES

The combined hardware/software technique that applies synchronization on data addresses is discussed into more detail in this section. We motivate why we opt for synchronization on data address instead of choosing for synchronization on instruction address.

The difference between the two synchronization methods is best explained by means of a small example. Consider the following instructions of a loop body

```
i0:  ...       # first instruction of loop body
i1:  load R1 R3    # R3 = Mem[ R1 ]
i2:  ...
i3:  incr R1   # increment index R1
i4:  jump i0   # next iteration
```

In this loop a stream is processed. Instruction `i1` loads the elements of the stream from memory. In case of synchronization on instruction address, there is an entry in a prefetch table that contains the instruction address (`i1`) and other information. Each time instruction `i1` is executed, it hits in the prefetch table which, in turn, may result in the generation of a prefetch. In case of synchronization on data address, the entry in the prefetch table contains the data address of the next stream element and other information. When a load operation is executed for this address, it hits in the prefetch table and, again, a prefetch may be generated. In addition, the field that contains the data address of the next element is updated by adding the stride. Next, we consider the same loop that is unrolled once

```
i0:  ...       # first instruction of loop body
i1:  load R1 R3    # R3 = Mem[ R1 ]
```

```
i2:  ...
i3:  incr R1   # increment index R1
i4:  ...
i5:  load R1 R3     # R3 = Mem[ R1 ]
i6:  ...
i7:  incr R1   # increment index R1
i8:  jump i0   # next iteration
```

Loop unrolling is one of the techniques that are applied in optimizing compilers for exposing instruction level parallelism. These kinds of code optimizations are an essential part of compilers for superscalar processors or VLIW processors, like the TriMedia TM1000 media processor [7]. Now, notice that there are two load operations (`i1` and `i5`) involved in fetching the elements of the stream: one load operation fetches the elements with an even index and the other load operation fetches the elements with an odd index. When synchronizing on instruction address, two table entries are needed for a single stream. In contrast, the technique that synchronizes on data addresses still needs one table entry. One can imagine what happens when loops are unrolled more than once. Heavy optimizations, either by code rewriting or by optimizing compilers, will typically be applied on critical inner loops. But, those are the loops that are expected to benefit from stream prefetching the most. Concluding, these observations indicate that it is more economical to implement synchronization on data address, since it requires a significantly smaller prefetch table than is required for synchronization on instruction address.

When implementing stream prefetching, the prefetch table will be a part of the data cache. Synchronization on instruction address requires the availability of the program counter for hit/miss detection in the prefetch table. Synchronization on data address, however, only needs information that is already available in the data cache.

As already mentioned in the previous section, it is far from trivial to determine the instruction address at the source code level. And, therefore, inserting stream prefetch instructions into the code by the programmer is considered a real burden. A compile-profile-compile approach may be needed here. Stream prefetch instructions with synchronization on data address can readily be applied, since the data address of a stream's first element is just a simple expression.

We have performed an evaluation [8] that shows that both synchronization techniques have similar performance results. For a more detailed architectural exploration we prefer synchronization on data address. This decision is based on the above observations that it will most likely be more economical to implement and that it is easier to apply without many compiler modifications.

5. BASIC HARDWARE ARCHITECTURE

In this section, we discuss how stream prefetching can be incorporated into the architecture of a data cache. Consider a non-blocking cache that has a number of *fetch units* that can fetch data from main memory (or a second level cache) independently from the operation of the CPU core. On a load miss, a fetch unit is initialized for the transfer of a cache block. When the last data of that cache block has been received by the cache, the fetch unit is released such that it can be initialized for a next transfer.

A cache that supports stream prefetch instructions will also support scalar prefetch instructions. The scalar prefetch is considered first. On a prefetch hit, nothing has to be done since the data is already available in the cache. On a prefetch miss, a fetch unit is initialized unless there is no free fetch unit left anymore for serving load misses. A fetch unit that handles a prefetch has a lower bus access priority than a fetch unit that handles a load miss since load misses stall the CPU while prefetch misses do not. In case a prefetch cannot initialize a fetch unit, the prefetch request is not cancelled but put into a *prefetch queue* such that it can be issued later in time. Having introduced a prefetch queue, it makes sense to redirect all prefetch misses to the queue first regardless the availability of fetch units. By doing so, prefetch instructions are processed on a first come first served basis. Whenever the CPU does not execute a cache operation and there is a fetch unit available for prefetches, a prefetch request is retrieved from the queue and processed by the cache (see Figure 1).

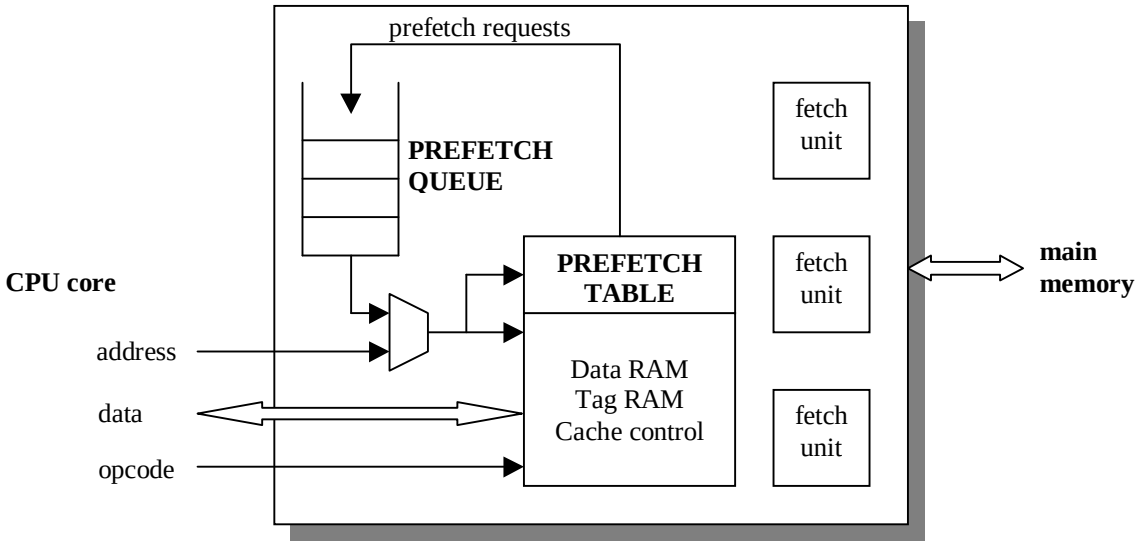

Figure 1 Data cache with prefetch queue and prefetch table

With respect to the prefetch queue it has to be decided what the queue length is and what to do on a queue overflow. A queue overflow occurs when more prefetch requests are generated than can be handled; there is not enough prefetch bandwidth. A simple solution is to discard the prefetch request that causes the overflow. This may lead to a cache miss later but will not change the functional correctness of the running application.

Next, we consider stream prefetch instructions. A stream prefetch instruction has 4 arguments: (i) *address* of the first data element, (ii) the *stride*, (iii) the *run ahead* that indicates how many data elements are to be prefetched in advance, and (iv) the *count*, i.e. the total number of stream elements. Processing a stream prefetch instruction results in initializing an entry in the prefetch table. An entry has the following fields: (i) *address* for synchronization, (ii) *stride*, (iii) *next prefetch address* for which a prefetch request is to be issued, and (iv) *count*, the number of stream elements left. At initialization, the next prefetch address is calculated from the address of the first data address, the run ahead value, and the stride. The count field also depends on the run ahead value. For example, a stream prefetch instruction with argument values

```
address = 0x00100000, stride = 4, run ahead = 16, count = 128
```

results in the initialization of a table entry with fields

```
address = 0x00100000, stride = 4, next prefetch address = 0x00100040, count = 112
```

where next prefetch address is the address of the 17^{th} stream element (0x00100000 + 16*4 = 0x00100040).

There are a number of ways to implement the initial prefetching of the first 'run ahead' (16 in the example) elements. We do not discuss those implementations here. We focus on the operation of the prefetch table once the initial prefetching has been completed.

On a load operation[2], the load address is compared with the addresses of the prefetch table, which is fully associative. On a miss, nothing is done. On a prefetch table hit, the count is decrement, a prefetch request for 'next prefetch address' may be sent to the prefetch queue, and the 'address' and 'next prefetch address' are both updated by adding the 'stride'. A prefetch request is issued only when it is a request for an address that is in a new cache block. Otherwise, multiple requests for the same cache block would cause unnecessary prefetch queue overflow. Notice that the operation of the prefetch table runs concurrently to normal cache control and therefore does not add overhead to the critical path.

[2] and store operation for caches that implement fetch-on-write

We conclude this section by pointing out the flexibility of the streamprefetch instruction. First, we show that it is possible to prevent the prefetch table from overflowing the prefetch queue with prefetch requests by choosing appropriate argument values. Next, we present how stream prefetching is applied in the context of multi-ported caches.

By simply choosing appropriate argument values for the streamprefetch instruction, one can prevent that the prefetch table generates too many prefetch requests for the same cache block. Consider the stream prefetch instruction mentioned before. The stream consists of 128 data elements with a stride of 4. Now, if the cache block size is 32 bytes, there will be 8 stream elements in one cache block. Thus, prefetching every 8^{th} element only also implements the prefetching of all stream elements. In the example, a cache with block size 32 bytes would initialize the prefetch table entry with fields

```
address = 0x00100000, stride = 32, next prefetch address = 0x00100060, count = 13
```

where a run ahead of 24 (3 times 8) elements is assumed.

A variation on the same theme provides a solution for the problem that discarding a prefetch request due to queue overflow results in a cache miss later, if the prefetch table issues only one request per cache block. A solution is to issue two prefetch requests per cache block, such that an occasional overflow can be caught up by a second request. Consider the same example again. For 32-byte cache blocks, prefetching every 4^{th} element results in the generation of two prefetch requests per cache block. The initialization of the prefetch table entry is

```
address = 0x00100000, stride = 16, next prefetch address = 0x00100050, count = 27
```

where the run ahead is 20 (5 times 4) elements.

We conclude by pointing out how stream prefetching is applied in the context of a dual ported cache. In such a cache, two cache operations can be executed in the same cycle. Assume that two successive data elements, s_i and s_{i+1}, are accessed in the same cycle. Hit/miss detection in the prefetch table may not be serialized since it is not known whether s_i or s_{i+1} is detected first. If s_i is detected first, the next address to synchronize on is the address of s_{i+1}, which is detected next. If, however, s_{i+1} is processed first, there will be no hit. After that s_i will hit the prefetch table and the next address to synchronize on is the address of s_{i+1}. But this element has already been accessed and, thus, no prefetches are issued ever again for this stream. Simultaneous hit/miss detection must result in a *double hit* such that the next address to synchronize on and the next prefetch address can be updated accordingly. Like before, a more elegant solution is possible by choosing the appropriate arguments for the stream prefetch instruction. For small strides, we choose to prefetch the substream that only contains the even indexed elements. This is done by doubling the stride and halving the count. For larger strides, we can execute two stream prefetch instructions, one for the even indexed elements and one for the odd indexed element.

6. EXPERIMENTAL RESULTS

We evaluated our stream prefetch technique by using a cycle accurate simulator of a research version of a TriMedia VLIW processor [9]. We added a stream prefetch instruction to the processor's instruction set and implemented support for this instruction in the C compiler. We also modified the cache simulator and implemented a prefetch table and a prefetch queue.

As a benchmark, we used two implementations of a video de-interlacing application that has been optimized to run on the TriMedia VLIW core [10]. This application processes video fields and its implementation typically contains a nested loop where the inner loop iterates over the pixels in a video line and where the outer loop iterates over the video lines in a video field. There are two positions in the code where stream prefetch instructions can be inserted, as is indicated in the following (abstract) code fragment.

```
/* A: stream prefetch instruction fetches entire field */
for ( line=0; line!=NROF_LINES; line++ )

  /* B: stream prefetch instruction fetches single line */
  for ( pix=0; pix!=NROF_PIXELS; pix++ )
   {
     process pixel;
   }
```

By inserting a stream prefetch instruction at position A, prefetching of an entire video field is initiated (referred to as *field prefetching*). By inserting a stream prefetch instruction at position B, each iteration of the outer loop initiates prefetching of a single video line (referred to as *line prefetching*).

An experiment is performed in the following way. Stream prefetch instructions with the proper argument values are inserted in the code on either position A or position B. The code is compiled for the target platform by an optimizing compiler and instruction scheduler. The resulting object code is run on a cycle accurate simulator that reports the following data for processing one video field:

- the number of executed cycles
- the number of prefetches that are retrieved from the prefetch queue
- the number of prefetches that result in a fetch from main memory (recall that a prefetch address from the queue that results in a cache hit does not result in a fetch from main memory)
- the number of load instructions that are executed
- the number of load misses

The data cache that is incorporated in the simulator implements a policy that allocates a cache block on a write-miss but does not perform a fetch on a write-miss (like the TriMedia TM-1000 does [7]). It is a 16 kBytes 8-way set-associative cache with a block size of 64 Bytes. There are 4 fetch units, but at most 2 of these fetch units may be used for prefetches. The prefetch table has 16 entries, but in our application only 3 are actually used.

Experiment 1

We investigated the effect of loop optimization on the effectiveness of stream prefetching. The additional parameters and test results are listed in Table 1. Unroll factor 2 means that the number of pixels that are processed in one loop iteration is doubled.

unroll factor	prefetching (yes/no)	# cycles	# prefetches executed	# prefetches from queue	load miss rate	reduction in cycle count
1	no	34834	-	-	4.14 %	-
1	yes	33313	234	5549	0.00 %	4.37 %
2	no	27385	-	-	4.14 %	-
2	yes	25862	234	5550	0.00 %	5.56 %
4	no	25204	-	-	4.14 %	-
4	yes	23683	234	3768	0.00 %	6.03 %

Table 1 Effectiveness of stream prefetching when applying loop unrolling
(field prefetching; pref.queue size 8; stride 4; run ahead 16)

Stream prefetching is able to eliminate all load misses. The table shows that the relative reduction in cycle count increases when loop optimizations are performed. This is readily explained from the fact that loop optimization leads to more efficient code whereas the miss penalties remain unchanged.

queue size	# cycles	# prefetches executed	# prefetches from queue	load miss rate
8	23683	234	3768	0.00 %
4	23683	234	1884	0.00 %
3	23683	234	1413	0.00 %
2	23684	235	942	0.00 %
1	24379	118	471	2.05 %

Table 2 Prefetch behavior related to prefetch queue size
(field prefetching; unroll factor 4; stride 4; run ahead 16)

Experiment 2

We investigated the influence of the prefetch queue size on the prefetch behavior. The results are listed in Table 2.

A prefetch queue of 2 entries is still large enough to eliminate all load misses for this application. The prefetch table generates 16 prefetch requests per block (stride of 4 in 64 B blocks). The total amount of generated prefetch requests is 5550. Obviously, many prefetch requests are cancelled due to prefetch queue overflow. The results show that enough relevant prefetch requests are stored in the queue unless the queue has only a single entry.

For the next experiments we assume that the target platform has instructions that exploit subword or SIMD parallelism. Assume that a data word contain pixel values of a number of pixels, say 4. An SIMD instruction operates on such 4-tuples in such a way that an operation is applied on each element of the 4-tuple individually, like in a 'quadadd' operation: quadadd.(a,b,c,d).(p,q,r,s) → (a+p,b+q,c+r,d+s).

Experiment 3

In this experiment, results are obtained from the implementation of our application in which SIMD operations are applied. The results are listed in Table 3.

prefetching (yes/no)	queue size	# cycles	# prefetches executed	# prefetches from queue	load miss rate	reduction in cycle count
no	-	11075	-	-	4.14 %	-
yes	8	9483	234	1474	0.00 %	14.37 %
yes	4	9471	234	1394	0.00 %	14.47 %
yes	2	9472	235	918	0.00 %	14.47 %
yes	1	10325	118	471	2.05 %	6.77 %

Table 3 Stream prefetching in an application that exploits SIMD parallelism
(field prefetching; unroll factor 4; stride 4; run ahead 16)

The results show that stream prefetching eliminates all load misses unless the queue size has only one entry. Comparing the 14.5 % cycle count reduction to the results in experiment 1 indicates that stream prefetching gains importance in the context of applications that use SIMD parallelism.

queue size	stride	run ahead	# cycles	# prefetches executed	# prefetches from queue	load miss rate
8	32	2	9769	217	741	0.42 %
8	64	1	9755	223	399	0.33 %
4	32	2	9776	216	679	0.44 %
4	64	1	9907	204	361	0.67 %
2	32	2	9828	208	454	0.51 %
2	64	1	10094	171	228	1.17 %
1	32	2	10393	114	228	2.15 %
1	64	1	10675	95	114	2.49 %

Table 4 Line prefetching with different strides
(SIMD version of application; unroll factor 4)

Experiment 4

This experiment uses line prefetching in stead of field prefetching. In addition, we used stream prefetch instruction with stride arguments that differ from 4. The results are listed in Table 4.

The results show that line prefetching has a worse performance (cycle count) than field prefetching (see experiment 3). A stride of 32 results in the generation of 2 prefetch requests per block by the prefetch table and a stride of 64 results in the generation of only 1 prefetch request. The results indicate that 2 requests per block have a better performance when many requests are discarded due to queue overflow. This is the case for small queue sizes (like 2 or 4).

Experiment 5

This experiment shows the effect of stream prefetching when the difference in processor speed and memory speed increases. Up to now, we assumed that it takes 8 bus cycles to transfer a 64 B cache block from memory. The performance gap between processor and memory is increased by taken 16 bus cycles and 24 bus cycles per block transfer. The results are listed in Table 5.

# bus cycles per transfer	prefetching (yes/no)	stride	run ahead	# cycles	# prefetches executed	# prefetches from queue	load miss rate	reduction in cycle count
8	no	-	-	11075	-	-	4.14 %	-
8	yes	64	1	9472	235	353	0.00 %	14.47 %
16	no	-	-	12805	-	-	4.14 %	-
16	yes	64	1	9538	235	353	0.00 %	25.51 %
24	no	-	-	15260	-	-	4.14 %	-
24	yes	64	1	12049	233	349	0.83 %	21.04 %

Table 5 Prefetch behavior in relation to memory speed
(SIMD version of application; queue size 8; field prefetching; unroll factor 4)

The results indicate that stream prefetching becomes more important when the performance gap between processor speed and memory speed increases. In case of 24 bus cycles per block transfer, the memory bandwidth is the limiting factor. Not all prefetches can be completed in time.

7. CONCLUDING REMARKS

In this paper, we presented a stream prefetch technique that synchronizes on data addresses. It is a combined hardware/software technique that uses a stream prefetch instruction to signal a hardware prefetch table to automatically prefetch a data stream. An important advantage of a combined hardware/software technique over a pure hardware technique is its cost-effective implementation. The main advantages over a pure software solution are that there is no instruction overhead in the loop body, that the hardware handles cache block alignment issues, and that other code optimization techniques do not interfere with prefetching.

An initial assessment shows similar performance results of our technique in comparison to a similar technique that synchronizes on instruction addresses. However, synchronization on data address has a number of advantages. First, at the source code level, the programmer has the information to apply stream prefetch instructions. A more advanced compile-profile-compile solution can be considered in the future. Secondly, compiler optimizations do not split data streams into numerous (sub)streams, as is the case with synchronization on instruction address. This implies that synchronization on data address only needs one prefetch table entry per data stream. And thirdly, synchronization on data address does not require the availability of the program counter in the data cache.

In Section 5, we discussed the implementation of stream prefetching. Assuming that a non-blocking cache that supports a scalar prefetch instruction already contains a prefetch queue, a fully associative prefetch table is needed to implement the stream prefetch technique. Since each data stream only requires one table entry, a table of 16 entries is expected to be large enough. Although we did not present data on the area complexity of such an implementation, we expect it to be cost effective. The implementation does not introduce overhead to the critical path.

Fine-tuning of the arguments of the stream prefetch instruction is an effective way of applying stream prefetching in multi-ported caches. The basic technique here is to enlarge the stride such that a substream, e.g. only the even indexed elements, is prefetched. In practice, i.e. considering small strides, this has the same effect as prefetching the entire stream. This technique can even be used for prefetching streams that have an irregular access pattern but contain a substream with a regular access pattern.

The performance results of Section 6 show the effectiveness of stream prefetching in eliminating cache misses. The introduction of SIMD instructions that process multiple data elements simultaneously will increase the fraction of memory instructions to be execution. In this context, prefetching, and stream prefetching in particular, gains importance. In addition, when the performance gap between processor speed and memory speed increases, the cycle count reduction increases significantly when stream prefetching is applied.

8. ACKNOWLEDGEMENTS

We like to thank all our colleagues in the PROMMPT project team who have contributed to the work that is presented in this paper.

9. REFERENCES

1. Norman P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers”, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 364-373, May 1990.
2. Subbarao Palacharla, and Richard. E. Kessler, “Evaluating stream buffers as a secondary cache replacement”, *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pp. 24-33, 1994.
3. Daniel F. Zucker, Michael J. Flynn, and Ruby B. Lee, “A comparison of hardware prefetching techniques for multimedia benchmarks”, *Proceedings of the International Conference on Multimedia Computing and Systems*, Hiroshima, Japan, June ’96.
4. Tien-Fu Chen, “An effective programmable prefetch engine for on-chip caches”, *Proceedings of MICRO-28*, pp. 237-242, IEEE Computer Society Press, 1995.
5. Daniel F. Zucker, Ruby B. Lee, and Michael J. Flynn, “An Automated Method for Software Controlled Cache Prefetching”, *Proceedings of the Thirty-first Hawaii International Conference on System Sciences*, Kona, Hawaii, January 1998.
6. Tien-Fu Chen, and Jean-Loup Baer, “Effective hardware-based data prefetching for high-performance processors”, *IEEE Transactions on Computers*, Vol.44, No.5, May 1995.
7. Gerrit A. Slavenburg, Selliah Rathnam, and Henk Dijkstra, “The TriMedia TM-1 PCI VLIW Media Processor”, *Proceedings of Hot Chips 8*, August 1996.
8. Andy D. Pimentel, Pieter Struik, and Pieter van der Wolf, “Evaluation of hardware-based and hybrid data prefetching techniques for the TM1”, internal report, Philips Research Laboratories Eindhoven, November 1997.
9. Frans Sijstermans, et.al., “Design space exploration for future TriMedia CPUs”, ICASSP 1998 (to appear)
10. Bram A.K. Riemens, Robert-Jan Schutten, and Kees A. Vissers, “High speed video de-interlacing with a programmable TriMedia VLIW core”, *Proceedings of ICASSP*, San Diego, CA, 1997.