Exploring DMA-assisted Prefetching Strategies for Software Caches on Multicore Clusters

Christian Pintò†, Luca Benini‡§
†DEI Department, University of Bologna, Italy - Email: {christian.pinto,luca.benini}@unibo.it
‡Integrated Systems Laboratory, ETH Zurich, Switzerland - Email: lbenini@iis.ee.ethz.ch

Abstract—Modern many-core programmable accelerators are often composed by several computing units grouped in clusters, with a shared per-cluster scratchpad data memory. The main programming challenge imposed by these architectures is to hide the external memory to on-chip scratchpad memory transfer latency, trying to overlap as much as possible memory transfers with actual computation. This problem is usually tackled using complex DMA-based programming patterns (e.g., double buffering), which require a heavy refactoring of applications. Software caches are an alternative to hand-optimized DMA programming. However, even if a software cache can reduce the programming effort, it is still relying on synchronous memory transfers. In fact in case of a cache miss, the new line is copied in cache and the requesting processor has to wait for the completion of the transfer. While waiting, processors are not able to perform any other computation. Cache lines prefetching can be used to reduce the number of synchronous memory transfers, and increase the active time of each processor, by loading cache lines before they are actually needed. In this work we explore various DMA-based prefetching techniques applied to a software cache implementation, presenting both automatic and programmer assisted prefetch mechanisms applied to computer vision kernels.

I. INTRODUCTION

Heterogeneous Multi-Processor Systems on a Chip (MPSoC) are a recent evolution in integrated computing platforms, where standard multi-core CPUs work alongside with highly parallel and at the same time energy efficient programmable acceleration fabrics [1] [2] [3]. Heterogeneous computing is motivated by the desire to lower the complexity of manufacturing the chip, and to follow the power consumption constraints imposed by the market. One example platform is STHORM of STMicroelectronics [4]. STHORM is composed by several computing units, packed in groups of 16 (Cluster). Processors in the same cluster have private I-Caches and share a multi-bank data scratchpad. The choice of having a shared data scratchpad inside each cluster makes it possible to achieve higher computational density (GOPS/mm2), as scratchpads are more area-efficient than caches [5]. Data scratchpads, however, force the programmer to deal explicitly with external-memory-to-scratchpad transfers. This is usually automatic, and transparent to the programmer, when a hardware data cache is available. Moreover, the gap between processors architectures and memories is growing, with the memory access latency being a plague for the performance of applications. Hiding this latency is thus a key factor when programming applications, especially for those with a high intensity of memory accesses.

A growing focus area for parallel memory-intensive applications is multimedia and computer vision, where computation is split into chunks and data subsets are repeatedly moved in and out from the external memory to the scratchpad. This type of computation is often subject to real-time constraints. It is immediate to understand how disruptive those external memory transfers can be, especially if synchronous (wait for the transfer to complete), on the global performance of an application. One well-known solution available today is overlapping memory transfers with computation. Application are structured to work on a block of data, while in the meantime the next block to be processed is being transferred into local memory using a Direct Memory Access (DMA) engine. This technique is called double-buffering.

DMA based programming with double-buffering has three main drawbacks: 1) applications need to be heavily modified to use double-buffering. 2) the available amount of local scratchpad can limit the effectiveness of the technique (two times the size of a block is needed). 3) the programmer has to deal with the trade-off between the size of the block to be transferred and the amount of computation performed on it. If the amount of work performed on each data block is not well chosen, it might not hide the transfer of the next data block.

An alternative to DMA programming which has always triggered interest are software caches. However, even if software caches can lower the programming complexity compared with DMA, they still have a major drawback: they are reactive. A software cache reacts according to the result of the lookup, and in case of miss a line refill is programmed. The completion of the transfer is waited before the application can continue. Processors waiting for the refill of a particular line can not perform any other task, wasting clock cycles. DMA-assisted prefetching can be used to anticipate the needs of processors by programming cache line transfers ahead of time, with the aim of minimizing the possibility of waiting for a certain datum. Software prefetching however is strongly dependent on the memory access pattern of applications: those with a regular access pattern can benefit from automatic prefetching techniques, prefetching cache lines according to a fixed rule. On the other hand, for applications with a more irregular access pattern, it is useful to exploit the hints of the programmer to know which line to prefetch next.

In this work is evaluated the effectiveness of DMA-assisted prefetching, applied to a parallel software cache implementation [6] for the STHORM acceleration fabric. The basic idea is to use DMA-prefetching to further minimize the number of blocking cache misses, and thus avoiding wait periods where processors waste clock cycles. The goal of this work is thus to transform the software cache in [6] into a pro-active entity. Both automatic and programmer assisted...
prefetching techniques have been evaluated when applied to three computer vision case studies: Viola-Jones Face Detection [7], Normalized Cross-Correlation (NCC) [8] and a color conversion algorithm. The overhead due to software prefetching is also discussed in the experimental results section.

The rest of the paper is organized as follows: in Section II we give an overview of related work on data prefetching and software caches. Sections III and IV briefly describe respectively the architecture targeted by this work, and the software cache taken into account. Section V describes the prefetching techniques used in this work. Section VI gives the implementation details of our prefetching infrastructure. Finally in Sections VII we show experimental results, and in Section VIII closing remarks and future work are presented.

II. RELATED WORK

Data prefetching is not new to the research community, first works appeared in the late 70s [9], and since then many other works have been published. The need for prefetching started with the beginning of processors performance boost, which at the end led to the multi-core era. Processors became faster and faster, while memory hierarchies were not able to follow the same trend. By using software prefetching it was possible to mitigate the effects of memory latency, and both hardware ([10], [11], [12], [13]) and software ([14], [15]) solutions have been studied. Hardware prefetching has the benefit of no extra code overhead, but the prefetch scheme is locked to what the underlying hardware permits. On the contrary, software based prefetching mechanisms allow to implement whatever prefetch one might be interested in, at the price of extra code execution due to address calculation and actual prefetch triggering. Both mechanisms of course are meant to solve the same problem: reduce the number of cache misses and mitigate the memory latency.

Also DMA engines have already been used for software prefetching: Authors in [16] present a DMA-based prefetch system which prefetches arrays with a high reuse rate, according to an analysis of the code aimed at minimizing the energy consumption and maximizing performance. DMA priorities and queues are used to resolve possible data dependencies between different iterations of the same kernel, ensuring the correctness of each memory access. Authors in [17] exploit the DMA engines available in the Cell BE processor, to prefetch data in I/O intensive workloads. Files are first read from the central core and then distributed to each SPE using DMA transfers.

We concentrate our attention on software prefetching applied to software caches for scratch-pad-based Clustered Multicore accelerators. Several works on software caches can be found in literature, with those related to scratch-pad based multicores mostly targeting the Cell BE processor. Examples are: [18], where authors propose an alternative software cache design aimed at reducing the miss rate. Authors in [19] present a software cache based OpenMP runtime for the Cell BE processor, leveraging the weak consistency model to ensure memory accesses correctness. Finally authors in [20] present a parallel software cache implementation for the STMicroelectronics STHORM acceleration fabric in which, differently from the previously described related work, the software cache is shared among all processors in a cluster.

In this work we want to mix software caching and DMA-based prefetching to further decrease the miss rate of the software cache runtime presented in [6], with the aim of minimizing the stall of processors waiting for a line to be refilled. Prefetching applied to software caches is not a new topic; authors in [20] implement a prefetching mechanism for irregular memory references, to reduce the miss rate of the software cache runtime provided with the Cell BE SSC Research Compiler. The approach is relying both on compiler support and run-time decisions. Authors in [20] apply prefetching only to irregular memory references, while regular ones are resolved using direct-buffering. In this work prefetching is applied to all memory accesses, tackling with the irregular ones by using the programmer assisted prefetching interface. In our approach regular references are still relying on software cache and prefetching, because techniques like direct-buffering reduce the amount of available local memory due to statically pre-allocated buffers.

III. TARGET MULTI-CORE ACCELERATOR

The target platform for this work is the ST Microelectronics STHORM many-core multi-cluster acceleration fabric, composed by computing clusters connected through an asynchronous NoC [4]. Each cluster (Fig. 1) features up to 16 STxP70 processors and a tightly coupled multi-banked data memory (TCDM) shared by the entire cluster. Processors access the TCDM through the Logarithmic Interconnect, which is a Mesh-of-Trees (MoT) allowing a 2 cycles access latency in absence of conflicts. The TCDM address space is not aliased making each TCDM memory visible by all clusters in the system.

Each STHORM cluster is also equipped with a DMA engine and two hardware features for intra-cluster synchronization and mutual exclusion support. Those facilities are useful to implement complex barrier schemes or just simple locks (mutexes).

![Fig. 1: STHORM Cluster architecture](image)

IV. SOFTWARE CACHE

This section briefly describes the software cache implementation used for this work, for a more detailed description please refer to [6].

A. Software cache data structures

The software cache is implemented as a set of data structures allocated on the L1 data memory (TCDM) of the cluster,
and thus shared between all the processing elements. Two tables are defined: the Tags table and the Lines table.

Each entry of the Tags table (32 bits wide) maintains the tag of the corresponding line and a dirty bit used for write-back operations. The Lines table is used to maintain the actual data of the cache, and is accessed by processing elements in case of cache hit.

B. Lookup function

The lookup routine is based on a hash function which extracts the index of the cache line associated to an address. The hash function used is very simple and is based on bitwise operations. Each time a lookup is executed, tag and cache line index are extracted from the address. The tag extracted from the address is compared with the one coming from the Tags table. If the lookup process ends with a miss the handling routine is called.

C. Concurrency management

Since the software cache taken into account for this work is shared among all processors in a cluster, it is necessary to regulate possible parallel accesses by different processors. To ensure correctness each line of the software cache is associated with a lock (mutex). Parallel accesses to the same line are regulated by the lock associated to the particular line, thus ensuring thread safety to each software cache access. Processors are allowed to access in parallel different lines of the cache, while concurrent accesses to the same line are serialized.

D. Operation modes

The software cache used in this work can be configured to work in two different modes: default mode and object-oriented mode. In the default mode each memory access is resolved with a cache lookup, regardless it is a byte, half-word or word access. In applications with a high frequency of cache accesses this can lead to a significant overhead.

To reduce the overhead, the object-oriented mode is used, where objects of fixed size are copied in cache at each lookup. The cache allows to access the entire object with a lookup only at the first access, further accesses to the same object can go directly in memory. We will use both modes in our experiments to highlight different features of the prefetching techniques we present.

V. PREFETCH TECHNIQUES

Even if a software cache can be used to heavily reduce the programming phase of an application, when compared to hand optimized DMA-based programming, it still suffers from one main limitation: a software cache is a reactive entity. When a processor, during the lookup phase, gets a miss it has to program the refill of the cache: victim selection, write back and finally the actual line refill. In this case the software cache is reacting to the miss event. During the refill phase, however, the requesting processor is stalled waiting for the new line to be copied in cache, and finally access it. When waiting for the line refill to complete, regardless of its implementation (DMA, explicit copies), processors are not allowed to perform any other work, wasting clock cycles. As already stated, the best way to get the full performance from a platform like STHORM is to hide as much as possible the latency of external-memory-to-scratchpad transfers, with the DMA playing a central role.

DMA-Assisted software prefetching is a good candidate to be used to further reduce the wait time of processors. Using line prefetching it is possible to anticipate the need of a processor, loading into the cache a line which will be accessed in the near future. However, the effectiveness of prefetching is strongly dependent from applications memory access pattern. Applications accessing the memory with a regular (predictable) pattern are likely to benefit from automatic prefetch mechanisms, prefetching cache lines with a fixed rule. While for applications where the access pattern depends on run-time data, it is preferable to rely on hints given by the programmer to trigger focused prefetch events. In this work we evaluate the effectiveness of software prefetching, exploring both automatic and programmer assisted prefetch schemes.

A. Automatic prefetch schemes

Many-core embedded accelerators are often used in the field of computer vision, where pipelines of algorithms are applied to frames coming from a video stream. Each stage of such vision applications is usually a parallel kernel scanning each frame, and applying the same algorithm to different sub-frames. Each sub-frame is in turn accessed with a fixed spatial pattern, depending on the coordinates of the pixel being processed at each time. In addition, most vision kernels are composed of nested loops, with the innermost accessing several contiguous memory locations. We implemented an automatic prefetch scheme, which at the first reference to a given line prefetches a new one close to the current. We decided to trigger the prefetch only at the first reference to a line, to follow the considerations made at the beginning of the section: the innermost loop of a vision kernel accesses contiguous data, likely to stay in the same cache line. So while computing on the current cache line, the next is prefetched. This strategy is meant to minimize the possibility of waiting for the completion of a prefetch, by exploiting the temporal locality of memory accesses.

![Fig. 2: Images access pattern](image_url)

Computer vision kernels are likely to access software cache lines with a certain spatial locality. Starting from the actual pixel coordinates, vision kernels usually access adjacent pixels in the same image row or in adjacent rows. Fig. 2 shows in different colors different threads, and their possible image access pattern (images are divided into cache lines). Two different prefetching policies are defined, called spatial prefetching policies. In the first policy the next adjacent line is prefetched...
at the first reference to each line (horizontal-prefetching), while in the second policy the cache line below (int the next row) the actual is prefetched (vertical-prefetching). To enable automatic prefetching the lookup routine of the software cache in [6] has been modified, and added few data structures needed to manage lines prefetching.

B. Programmer assisted prefetch

As already stated, automatic prefetching may be less effective when the access pattern of the application is not predictable, or does not follow the automatic prefetch policy. We have then implemented two programmer-assisted prefetch techniques, based on the following assumptions. Several vision applications access memory buffers computing the address using runtime data. Automatic prefetch may suffer of a non-predictable memory access pattern, we have then defined an API which allows the programmer to manually trigger the prefetch of the next line used by the application. The first programmer assisted prefetch API is composed by the following function:

```c
void prefetch_line (void *ext_mem_address, cache_parameters *cache);
```

This function programs the prefetch of the cache line where the address `ext_mem_address` is contained. The second parameter (cache) holds the pointer to the data structure maintaining the whole software cache status. The completion of the DMA transfer triggered for each prefetch is checked only at the first reference to the line.

The second manual prefetch interface can be used to avoid cold misses. Some applications access data buffers starting from the first elements through its end, examples are arrays or trees. Almost all applications have an initialization phase, in which indexes and addresses needed to share the computation among multiple processors are computed (prolog). It is possible to exploit the prolog of an application to prefetch the content of the whole cache, or part of it, with the aim of avoiding cold misses (initial cache prefetching). The programming API is composed by two functions:

```c
dma_req_t *prefetch_cache (void *ext_mem_address, uint32_t from);

void wait_cache_prefetch (dma_req_t *dma_events);
```

The first function (prefetch_cache) takes two input parameters: `ext_mem_address` represents the address in global memory of the buffer to prefetch, while `from` represents the starting offset in byte from which the prefetch will start. We decided to put the `from` parameter, to give programmers the freedom to start the prefetch from the point in the data buffer they consider to be the best (i.e. the computation starts accessing from there). The value returned holds the pointer to the DMA event associated with the cache prefetch. The second function (wait_cache_prefetch) is used to wait for the completion of the prefetch. The only input parameter is `dma_events`, used to provide the DMA event pointer to wait for (returned by `wait_cache_prefetch`).

wait_cache_prefetch is usually placed just before the beginning of the portion of the application using the software cache.

VI. Prefetch infrastructure

In this section are described the modifications made to the software cache in [6], to support both automatic and programmer assisted prefetch. The part mainly involved by modifications is the lookup routine, but also some additional data structures have been defined.

A. Additional data structures

Each prefetch event is associated to a DMA event generated by the DMA API itself. A prefetch is usually triggered by a core while its completion is likely to be verified by another one. To support the sharing of DMA events we defined a common buffer of 16 events (events_buf), making the assumption of a maximum of one prefetch per core active at the same time.

The status of each cache line has been extended with two extra flags: the PREFETCHING flag and the FIRST_ACCESS flag. The former is used to check if a line is prefetching, while the latter is used to check for the first access to a cache line. The usage of the two flags is better explained later in this section. Due to the need of the two new flags, also the dirty bit has been moved from the tag of the line to a new line status field (1 byte). Each cache line has also a private memory location, containing the index of the DMA event associated to its prefetch (1 byte). The index of the event is obtained when initializing the prefetch and is computed using events_buf as a circular buffer. The circular buffer uses a mutex (T&S based), to avoid different processors accessing the DMA events buffer at the same time.

To summarize, with respect to the original implementation we add: 2 bytes per cache line, a buffer common to the entire cache composed by 16 DMA events (32 bytes each), and an extra lock used to manage the DMA events buffer.

B. Lookup routine extension

The lookup routine has been modified in order to support line prefetch. Since the prefetch system presented in this work is based only on software, it adds some overhead to the original software cache implementation. Such overhead, only due to the extra code, will be discussed in Section VII.

At each cache lookup it is first verified if the current line is already being prefetched, by checking the PREFETCHING flag in the line status register. Processors trying to access a line which is already prefetching are forced to wait, until the DMA transfer is completed. The DMA event to wait for can be easily found using the private location associated to the line. Once the prefetch is finished, the normal lookup phase is executed: check if the current access is a hit or a miss.

The final phase of the lookup checks if the line is being referenced for the first time, and if this is the case the prefetch of a new line is programmed. We decided to trigger the prefetch at the first reference to a cache line, to maximize the possible overlap between the computation on the actual line and the prefetch of the next one. The line to be prefetched is chosen according to the prefetch policy being used.
C. Line prefetch subroutine

The line prefetch subroutine is in charge of applying the prefetch policy, according to the line which is currently accessed or to the hint of the programmer. The first important step performed in this phase is to identify the prefetch candidate, and acquire the lock of the line in which it will be placed (prefetch victim, identified according to the associativity policy of the software cache). This is necessary to avoid overwriting a line which is being accessed by some other processor. Before actually programming the DMA transfer a set of sanity checks is performed, to understand whether the prefetch should be triggered or not:

1) The prefetch victim is in turn prefetching: If the victim selected to be substituted by the new line is in turn prefetching, the processor must wait for the completion of the prefetch.

2) Presence in cache of the line to prefetch: If the line to be prefetched is already in cache, there is no need to prefetch it.

The two conditions may be overlapping (i.e. the line a processor wants to prefetch is already prefetching), in that particular case the prefetch routine will not modify at all the status of the software cache. This is likely to be true when two or more processors try to prefetch the same line.

Once the lock is acquired and all sanity checks are passed, the actual prefetch is triggered. The prefetch is essentially composed by three phases: 1) DMA event index acquisition, 2) source and destination addresses calculation, 3) DMA transfer programming. After the DMA transfer is programmed the index of the DMA event is saved into the specific location associated with the line, making it available to other processors. The first of the three phases accesses the circular buffer event_buf, to get the first free event slot. That phase requires the processor to acquire a lock, before changing the status of the buffer.

VII. EXPERIMENTAL RESULTS

A. Experimental Setup

All our experiments have been performed on a real implementation of the STHORM acceleration fabric in 28 nm CMOS. The evaluation board consists of a Xilinx ZYNQ ZC-702 chip (Host) featuring a dual Core ARM Cortex A9, mainly used to run the Android operating system and to submit tasks to the STHORM fabric (using OpenCL). The STHORM chip is composed by four computing clusters, each with 16 STxP70 processors, working at a clock frequency of 430 MHz, and a shared data scratchpad of 256 KBytes. The memory bandwidth towards the external memory is ∼ 300 MB/sec. The bandwidth is limited by a slow link between STHORM and the host system, implemented on the FPGA inside the ZYNQ chip.

All the benchmarks are implemented in OpenCL. The OpenCL runtime used is part of the STHORM SDK provided together with the evaluation board. The software cache and all prefetching extensions are implemented in standard C code, which is compiled and linked as an external library to the application. Results are presented in terms of execution time (nano seconds) or clock cycles, the latter are extracted using the hardware counters available in the STHORM chip.

In this section we first characterize the overhead due to prefetching, and then apply it to three computer vision case-studies, namely: Normalized Cross-Correlation (NCC) [8], Viola-Jones Face Detection [7], and a color conversion algorithm. The three use cases show how prefetching may be used to mitigate effects due both to the implementation of the software cache, and to the characteristics of the benchmark.

B. Prefetching Overhead Characterization

In this section we characterize the overhead added to the lookup routine by the prefetching infrastructure. Results are shown both in terms of executed instructions, and execution clock cycles overhead. To highlight what related only to prefetching, this experiment has been done on a single core run, avoiding any extra overhead due to contention amongst different processors. Numbers are extracted from the hardware counters available in the STxP70 processor.

In Table I we show the comparison of the number of instructions and clock cycles of the lookup routine, when prefetching is enabled and when disabled. The critical path of the software cache is the lookup&hit, which in its default implementation takes 11 instructions for a total of 18 clock cycles. While a miss takes 118 instructions and 893 clock cycles (line size of 32 bytes).

When prefetching is enabled three cases are possible: Hit, Hit&Prefetch and Miss&Prefetch. Note that the miss only case is not considered because prefetching is triggered at the first access to a line, as in the case of a miss.

In case of Hit the instructions count is increased to 21, and the clock cycles count to 46. We noticed that the ratio between the number of instructions and clock cycles (IPC) is not the same as in the case without prefetching, but it is lower. To better understand, we have investigated the STxP70 assembly implementation of the lookup procedure when prefetching is enabled. The reduction of the IPC is mainly due to two pipeline flushes, introduced by two conditional instructions: the first is generated when checking if the line is already prefetching, while the second is generated when checking if the line is accessed for the first time. In both cases the flush is introduced when the check has a negative result: line not prefetching or not first access. In case of Hit both conditions evaluate false, and the two pipeline flushes take place. The two conditional constructs are part of the prefetch infrastructure, and are generated by the STxP70 C compiler.

The Hit&Prefetch case is verified in case of hit and first access to the line. In this case the instruction count is 127, and the clock cycles spent are 197. Here it is possible to understand the effectiveness of prefetching: when prefetching, in fact, the instruction count of a hit (127 instructions) is close to the one in case of cache miss without prefetching (118 instructions). The main difference is in the number of clock cycles: in the former case the clock cycles count is much lower, 197 clock cycles, against the 893 of the blocking cache miss case. The prefetch of a cache line triggers an asynchronous DMA transfer, which overlaps with the execution of the processor. The first access to the prefetched line will be a hit, with the requesting processor saving ∼ 700 clock cycles. The saved cycles would be otherwise spent in waiting for the line refill.

In Table II we show the comparison between the cache miss case and the prefetched case, for different instructions count. The results show the overhead added by the prefetching infrastructure, and how prefetching reduces the number of clock cycles by almost half. The overhead is particularly high for the Hit&Prefetch case, which requires two pipeline flushes, one when checking if the line is already prefetching, and then checking if the line is accessed for the first time.
The last case is the Miss\&prefetch, in which the instruction count is increased to 215 with 1073 clock cycles spent in total. This case is the one which is less suffering the code increase overhead, because the miss handling routine is already composed by several instructions.

C. Case study 1: Normalized Cross Correlation

Normalized cross correlation (NCC) is a kernel used in computer vision, in the context of video surveillance [8] [21]. A typical application is in security systems for airports or public access buildings, where abandoned objects may be dangerous for the people inside the building. NCC works on two images: the background and the foreground. The background image is a static scene, taken when the ambient is free from any unwanted object. The foreground is acquired periodically from a camera, and compared with the background. NCC compares the two frames identifying any removed/abandoned object. The algorithm has been parallelized to be executed on multiple clusters, with each cluster working on a subset of the rows of the whole image. Inside each cluster the work is parallelized again in a column wise way, where processors access pixels belonging to the same column (or a subset of adjacent columns). We applied to this benchmark all the prefetching techniques discussed, expecting the vertical-prefetching to perform the best. Two software caches are used to access the background and foreground frames, each of them of 32 KBytes. The line size has been used as a parameter of the experiment, varying from 32 to 128 bytes.

Fig. 3 shows the execution time of the application in all the configurations. The DMA series (considered as the baseline) in the chart refers to a hand optimized version of the benchmark, using DMA double buffering instead of software caches. When the software cache is used without prefetching, the slowdown with respect to the DMA hand-tuned implementation is of \( \sim 40\% \), with a line size of 128 bytes. The penalty is that high because the innermost loop of NCC is tight, with just 3 arithmetic operations per cache access. The computation in this case is not able to hide the intrinsic overhead of the software cache. We then use prefetching trying to mitigate such overhead.

It is immediately visible that the vertical-prefetching technique is the best performing of the three, bringing the software cache implementation performance closer to the DMA hand-tuned version with a slowdown of only 5 %. The other two techniques horizontal-prefetching and initial cache prefetching are less performing, but still reduce the overhead respectively to \( \sim 10\% \) and \( \sim 24\% \). The initial cache prefetching has a poor performance because the prologue of the benchmark is not long enough to hide the prefetching of the cache. To summarize, the overall benefit due to prefetching is shown in figure Fig. 4, for all prefetching techniques and lines size. For this benchmark we measured an increase in the number of executed instructions ranging from 19% to 24%. Despite that, automatic prefetching is still beneficial for the overall performance of the application.

D. Case study 2: Face Detection

The second case study used for this work is Viola-Jones Face Detection [7]. The Viola-Jones face detection algorithm is based on a cascade classifier, that working on the integral image of the original picture is able to recognize if a subregion of an image contains a face or not. We decided to cache the cascade classifier. For this experiment we used the object-caching mode available in the software cache in [6]: at each cache miss an entire descriptor is copied in cache. Processors access only the first byte of the object through the cache, while further accesses to the same object do not involve the software cache runtime. This is meant to reduce the overhead of the software cache, due to a high number of cache accesses. The size of the object is set to 32 bytes, as the size of the cascade descriptor, and the total size of the software cache is set to 64 KB.

With this benchmark we want to check if prefetching is able to mitigate effects due to the specific algorithm or data set. Fig. 5 shows the execution time of the face detection benchmark, normalized to the DMA hand optimized version. As it is immediately visible, for image1, when no prefetching is used the performance is quite far from the DMA optimized version. While this is not happening for image2. This behavior is due to the size of the image, and the miss rate associated to it: image1 in fact is a very small image (43x52) with a total of 5 faces and 7.5% miss rate. The computation is thus not long enough to hide the overhead introduced by the software cache.

When applying prefetching, the overhead of the software cache is heavily reduced, and the slowdown is reduced from \( 8.6 \times \) to \( 2.5 \times \). This effect is less visible for image2 because the total number of misses is hidden by the total number of cache accesses, much higher than that in the previous case (image2 is 200 x 128 pixels, with a total of four faces). To better understand: the number of misses for image1 is 2483 and goes down to 435 when prefetching is applied, over a total of 3317 cache accesses. The number of cache misses measured for image2 drops from 2782 to 1044, over a total of 405316 cache accesses. Fig. 6 shows the overall miss reduction percentage. The initial cache prefetching is much more effective than line prefetching because Face Detection has a long prologue, and the initial DMA prefetch is completely hidden.

E. Case Study 3: Color Conversion

The last case study presented is a color conversion algorithm. In this application the input image is an RGB frame, and the output is the grey-scale conversion of the input. In this case study the image is divided in rows, and groups of rows are assigned to different processors. The characteristic of this benchmark is that there is no re-usage of the input data.

<table>
<thead>
<tr>
<th></th>
<th>Hit</th>
<th>Miss</th>
<th>Hit&amp;Prefetch</th>
<th>Miss&amp;Prefetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>No prefetch</td>
<td>11/18</td>
<td>11/8/953</td>
<td>127/197</td>
<td>213/1073</td>
</tr>
<tr>
<td>Prefetch</td>
<td>21/246</td>
<td>21/7/124</td>
<td>21/46</td>
<td>21/46</td>
</tr>
</tbody>
</table>

TABLE I: Prefetch overhead added to the lookup function, each cell contains: \#instructions / \#clock cycles (line size 32 bytes)
Fig. 3: NCC Benchmark execution time

Fig. 4: NCC Improvement due to prefetching with respect to the software cache without prefetching

Fig. 5: Face Detection execution time normalized to a DMA hand-tuned implementation

Fig. 6: Face Detection miss reduction percentage

Fig. 7: Color conversion execution time normalized to DMA hand-tuned implementation

The access pattern of the application.

As visible in figure, the reactive software cache has an average slowdown of 35%, when compared to the DMA implementation. With such a slowdown it may not be worthwhile to use the software cache, but instead to implement a double buffering system. Enabling line prefetching the overall slowdown is reduced, reaching the 13% in the best case. This slowdown value is an acceptable compromise between performance and ease of programming. As expected line prefetching can hide the cache refill of each line, and even in absence of data re-use it can significantly improve the performance of the application.

VIII. Conclusions

In this work we made a preliminary evaluation of DMA-assisted prefetching, applied to a software-cache for scratchpad based multi-core acceleration fabrics [6]. The aim of the work is to evaluate whether prefetching can be beneficial in further improving the software cache runtime. Not all applications will benefit from automatic prefetching, we have thus presented a set of programmer-assisted prefetching techniques. The schemes presented have been designed taking into account the typical memory access pattern of computer vision applications, in which spatial locality of accesses is often present and should be exploited. With our experimental results we tried to understand if prefetching is able to mitigate effects due to both the software cache runtime itself, or the application/dataset. Results show that prefetching is a promising optimization, which allowed us to finely optimize our benchmarks to reach a performance very close to the DMA hand optimized version. This is the case of NCC, where the overall slowdown with respect to the hand-tuned version is only 4%. The second
benchmark instead allowed us to see that even in case of non conventional datasets (image1, small image and high miss rate), prefetching can heavily reduce the overhead of the software cache speeding-up the execution of $\sim 3.5 \times$. With a global reduction in the number of cache misses up-to 86\%. In the last case study presented (color conversion), we prove that prefetching may be useful even in cases where software caching is less powerful (e.g. for applications with a low data re-usage rate). The overall slowdown is reduced to the 13\% in the best case.

Prefetching demonstrates to be a powerful optimization even in the context of software caches, and should be studied more in detail. Possible future optimizations focus on compiler assistance, to automate the usage of prefetch even in case of irregular memory access patterns.

ACKNOWLEDGEMENTS

This work has been supported by the EU FP7 projects vrltical (contract number: 288574) and P-SOCRATES (contract number: 611016).

REFERENCES


