SAD Prefetching for MPEG4 Using Flux Caches

Georgi N. Gaydadjiev and Stamatis Vassiliadis

Computer Engineering Laboratory,
Electrical Engineering, Mathematics and Computer Science Dept.,
EEMCS, TU Delft, The Netherlands
`G.N.Gaydadjiev, S.Vassiliadis@ewi.tudelft.nl`
`http://ce.et.tudelft.nl`

Abstract. In this paper, we consider flux caches prefetching and a media application. We analyze the MPEG4 encoder workload with realistic data set in a scenario representative for the embedded systems domain. Our study shows that different well known data prefetch mechanisms can gain little reduction in the cache miss ratios when applied on the complete MPEG4 application. Furthermore, we investigate the potential improvement when dedicated prefetching strategies are applied to the sum of absolute differences (SAD) kernels in MPEG4. We propose a flux cache mechanism that dynamically invokes cache designs with dedicated prefetching engines that can fully utilize the available memory bandwidth. We show that our proposal improves the cache miss ratios by a factor close to 3x.

Keywords: Flux caches, Prefetching mechanisms, Reconfigurable architectures, Multimedia.

1 Introduction

Flux caches [1] have been proposed as a microarchitectural alternative hardware mechanism for improving the performance of memories when compared to the hardwired caches. They are based on two main assumptions. The first assumption regards the availability of technologies that can be reconfigured before and/or during program execution. The second assumption regards the changing memory access behavior from program to program and during program execution. Flux caches are cache hierarchy designs that change dynamically their hardware organization to capture the memory access requirements of a given program/program execution. Flux caches assume implicit and explicit dynamic cache calls to "redesign and place" new hardwired caches instead of having permanent and unchangeable caches. In order to establish the validity of the approach, this paper assumes a real application and considers prefetching- one of the cache design aspects. Our experiments with some well know prefetching mechanisms and dynamic execution suggest that the above conjunctures hold true. For the mechanisms considered we also provide guidelines of how to design a Flux cache for sum of absolute differences (SAD) prefetching in MPEG4. It is noted that our investigation is not intended to propose a novel cache prefetching mechanisms but it is rather focusing on cache adaptation to "fit" the application behavior.

The contributions of this paper can be summarized by the following:

– A careful investigation of the memory behavior of a real-life MPEG4 encoding application working on a representative workload;
– Identification of potential kernels that can benefit from a dedicated prefetch method;
– Special Flux cache organization to fully utilize the available main memory bandwidth;
– Improvement of the cache miss ratios by a factor close to 3x.

The rest of this paper is organized as follows: Section 2 briefly reviews the most relevant related work on prefetching. Section 3 describes our experimental methodology and introduces the MPEG4 flux cache design for optimal prefetching of SAD8 and SAD16 data. In Section 4 the performance results that support our idea are described. Finally, the discussion is concluded in Section 5.

2 Background

In this section we will provide only the background information needed directly in the reminder of this paper. It introduces the two major parts used in our work: the flux cache concept and a very brief classification of cache data prefetch mechanisms. This is mainly due to space limitations and the fact that cache prefetching techniques have been a topic of research for numerous years. The interested reader can refer to overview papers such as [2, 3] where an elaborated discussion is presented. In addition, due to the subject we investigate, our background information on cache prefetching will be limited to the data cache prefetching only.

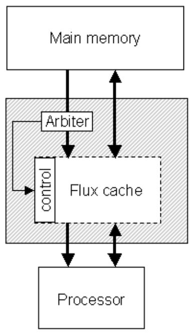

Fig. 1. Flux cache

Flux caches: Flux caches are fully customizable memory levels, envisioned for reconfigurable hardware implementation, that can be instantiated on demand. The flux cache reconfiguration can be performed before or during program execution. Implementations of arbitrary hardware cache design can be "programmed" under software or hardware control at runtime. The specific flux cache implementations are pre-determined at hardware/software co-design stage, i.e. by using application partitioning, monitoring, profiling or else. In its general form flux caches would require additional ISA support, however it has being shown that this is not always necessary, e.g. as in the MOLEN [4] polymorphic processor case (see [1]). The reconfiguration of the intended cache design is expected to introduce some reconfiguration overhead, although the benefits of using the flux cache during program execution are likely to compensate for it. The general flux cache organization is depicted in Figure 1.

The three main components of a flux cache are: the arbiter, the control unit and the reconfigurable HW area available for different cache instantiations. The flux cache parameters are usually implicit, however explicit calls (as it will be shown later in this text) can also be used. There is essentially a single *put* phase initiated by the arbiter after detection of a **put** instruction that interrupts the processor program flow until the hardware configuration is completed. The **put** instruction will be redirected to the control unit and interpreted accordingly. More precisely, the configuration microcode located at the targeted address will be loaded into the configuration memory to ensure the flux cache hardware structure. After the cache reconfiguration is completed (and all *valid* tags of the "new" cache are invalidated) the execution of the processor will continue with the execution of the next instruction keeping the processor execution consistency intact.

Prefetching for Data Caches classification: Prefetching has been extensively considered to improve cache performance. Data cache prefetching mechanisms the subject of our investigation can be divided into three major classes: *Software*, *Hardware* and *Hybrid HW/SW* data prefetching schemes. The software data prefetching has the potential of issuing requests for only the data that is expected to be used. This is due to the ability to have an application wide (compiler) view. The major drawbacks of the software data prefetching are the additional prefetch instructions overhead, the inability of the compiler to estimate run-time cache miss latencies and difficulty with prefetching of addresses unknown at compile time, e.g. pointer references. The hardware approaches have a zero processor overhead and direct access to run-time (latency) information. In addition, it has been shown that problems inherent to hardware prefetching such as memory bandwidth contention and cache pollution can be addressed effectively [5]. On the other hand, the hardware has no direct knowledge of future references and usually operates within a very limited scope. Furthermore, the hardware based mechanisms always trade between accuracy and coverage and can only exploit structured data access patterns. As expected, the hybrid hardware/software data prefetching approaches gain in popularity lately. A variety of schemes have been proposed ranging from prefetching of very specific access patterns to more generally applicable approaches [6, 7, 8, 9]. It should be noted that work in this area is often focused on multiprocessor machines.

The application of the flux caches for prefetching can benefit from the advantages of both the software and the hardware approaches. More precisely they can combine the compiler knowledge on references far ahead of the SW techniques with the zero processor overhead and run-time latencies awareness of the hardware approaches. Because flux caches will be called on demand (dynamically) multiple HW/SW schemes can coexist for single program execution. In addition, in respect to the proposed hybrid approaches, our proposal do not involve any additional ISA extension (see [1] for more information) and can instantiate any of the previously proposed schemes. In the next section we will show how specific memory intensive MPEG4 kernels can benefit form our approach. It should be noted that although here we focus on regular array accesses only, similar schemes can be applied for speeding up memory access to complex pointer structures (i.e. recursive pointers), sparse matrices or for support of novel vectorization mechanisms [10, 11].

3 MPEG4 Encoder Prefetching Investigation

In this section, before focusing on the envisioned solution we provide discussion of the methodology we used to perform our evaluation.

Methodology: We base our study on memory traces and the dinero IV [12] trace driven cache simulator for L1 and L2. Although the traces used in this study became rather huge (hundreds of gigabytes), we did not have the means to perform direct hardware measurements - widely accepted as the second preferred method for cache performance evaluation [13]. The dinero style application traces for this study where obtained using a modified in-order SimpleScalar 4.0 simulator [14].

As our benchmark we selected a complete MPEG4 encoder application and a set of representative workloads. This in contrast with the widely used Mediabench and EEMBC benchmark suites that concentrate on small kernels and limited datasets. In order to avoid library calls overhead we created a single statically linked executable based on xvidcore v.1.1.0 library and xvid_encraw.c raw format MPEG4 encoder. As dataloads we used five of the widely used video conferencing test sequences: *foreman*, *carphone*, *claire*, *miss america* and *grandma*. We obtained those from the Stanford Center for Image Systems Engineering web site [15]. All video sequences used are in raw format, YUV concatenated with sub-sampled UV components. The image dimensions are 144 lines x 176 pixels per line (or *Quarter Common Intermediate Format* (QCIF)) with 30 frames per second as specified by ITU H.261 video conferencing standard [16]. The sequence lengths are: 400 (for foreman), 382 (carphone), 494 (claire), 150 (miss america) and 870 (grandma) frames respectively. The produced MPEG4 output is in ”raw” format m4v that was found sufficient for our study. We validated the correctness of the compressed output by using *ffmpeg* [17]. We limited our study to only five out of nine available sequences after we found that the data loads play a minor role for the data miss ratios. As an example, the miss ratios found vary from 0.26 (foreman) to 0.32 (claire) for exactly the same 2k direct mapped data cache configuration. This is why only the best performing *foreman.qcif* encoding scenario will be considered in this study. This to investigate our proposal under worst case conditions.

For our experimental cache we use 2k split instruction and data direct mapped cache with 16 byte lines with no sub-blocks. This in attempt to evaluate the effects of the proposed mechanisms instead of working at the noise levels. In addition, we aim at a solution for simple embedded processors (without any SIMD extensions) and MPEG4 encoding. The findings in the text hereafter are general and will show similar relative improvements on arbitrary chosen realistic data cache sizes, applications and workloads.

Performance Evaluation: To identify the potential targets that may benefit from prefetching ”on demand” we used both code execution profiling as well as memory trace analysis. Profiling results show that SAD8 and SAD16 are responsible for 35.58% and 9.07% of the application cumulative execution time. In addition, the memory trace analysis indicated that 39.7% of the total data

Table 1. MPEG4 encoder cache miss ratios and memory fetches

	miss ratios			**fetches**	
	demand	prefetch	**total**	prefetch	**total**
always	0.265	0.2501	0.2582	1,754,619,345	*3,822,878,836*
load	0.2682	0	0.2682	0	2,068,259,491
miss	0.2649	0.8399	0.3717	471,670,789	2,539,930,280
OBL[a]	0.2682	0	*0.1451*	1,754,619,345	*3,822,878,836*
tagged	0.2645	0.84	0.3756	494,668,112	2,562,927,603
on demand	0.2682	-	0.2682	-	2,068,259,491

[a] *in dinero VI prefetch distance 1 with sub-block placement disabled*

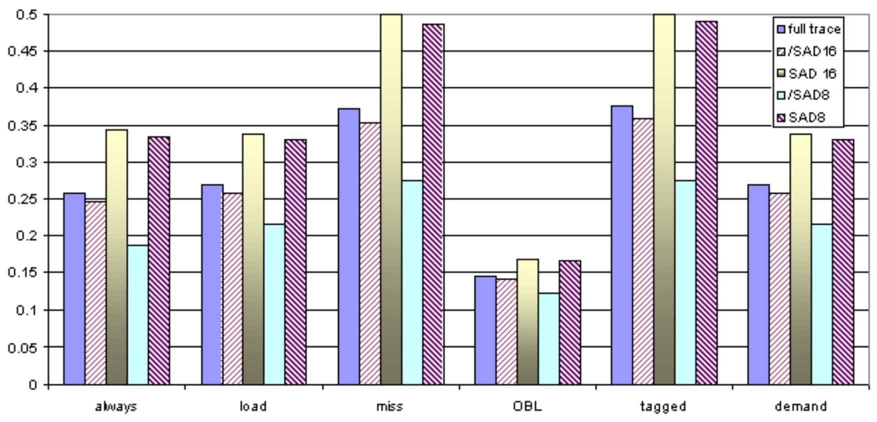

Fig. 2. SAD8 and SAD16 cache impact

memory reads are due to SAD8 (and 11.06% for SAD16) that makes both kernels primary candidates for prefetch optimizations.

We first started with evaluation of the existing prefetch techniques implemented in dinero IV. Table 1 depicts the miss ratios of a 2k/16bytes direct mapped data cache for our executable and video sequence (MPEG4 encoder and foreman.qcif). This table shows that the traditional prefetch strategies do not have significant impact on the miss ratios for the considered workload. The only strategy that shows some improvement (reduction from 26.8 down to 14.5 %) is the *one block look ahead* (OBL). This scheme initiates a prefetch of block $a + 1$ when block a is accessed. Such behavior fits well with the memory access patterns of the investigated MPEG4 kernels as will be shown later in this paper. A major drawback of OBL (as in the case with always prefetch) is the doubled number of main memory accesses compared to the ”no-prefetch” scheme presented in the last row of Table 1. The always prefetch (25.8%) performs very similar to a cache design without any prefetch (26.8%), while the prefetch on miss (37.2%) and the tagged prefetch (37.5%) show degradation in the miss ratios for this particular application. The prefetch on load (26.8%) is applying essentially the demand fetch policy (no prefetch) since we have not defined any sub-blocks in our experimental data cache. The latter fact will cause disabling of the sub-block placement and respectively the load-forward-prefetch as explained in the dinero IV documentation.

Next, we produced partial memory traces to investigate the relative behavior of the SAD8 and SAD16 kernels compared to the "reminder" of the application code. Again, we based out experiments on the same cache organization and size for the sake of a common reference for comparison. The main question was how the different parts of the investigated workload will influence the cache performance. The results are summarized in Figure 2. As it can be seen on this figure, the identified kernels perform worse than the full application when using the same cache size. In general, the "overall" code (/SAD16, read as *not* SAD16, and /SAD8 on Figure 2) shows a very minimal improvement of a couple of percents compared to the complete application figures. Considering the significant contribution of both kernels to the total number of MPEG4 encoder memory reads, we took a closer look at their internal structure.

```
uint32_t sad8_c(const uint8_t * const cur,
       const uint8_t * const ref,
       const uint32_t stride)
{
    uint32_t sad = 0;
    uint32_t j;
    uint8_t const *ptr_cur = cur;
    uint8_t const *ptr_ref = ref;
    for (j = 0; j < 8; j++) {
        sad += abs(ptr_cur[0] - ptr_ref[0]);
        sad += abs(ptr_cur[1] - ptr_ref[1]);
        sad += abs(ptr_cur[2] - ptr_ref[2]);
        sad += abs(ptr_cur[3] - ptr_ref[3]);
        sad += abs(ptr_cur[4] - ptr_ref[4]);
        sad += abs(ptr_cur[5] - ptr_ref[5]);
        sad += abs(ptr_cur[6] - ptr_ref[6]);
        sad += abs(ptr_cur[7] - ptr_ref[7]);
        ptr_cur += stride;
        ptr_ref += stride;
    }
    return sad;
}
```

Fig. 3. SAD8 C code

The C-code of the SAD8 kernel is shown on Figure 3. SAD16 loop has a similar structure with twice as long body and doubled number of iterations. As it can be seen the memory accesses of the SAD operation are predominately reads of array elements. In addition, the memory access pattern of the complete loop is 100% deterministic and is basically predefined by the three input parameters. This is an advantage that should be exploited.

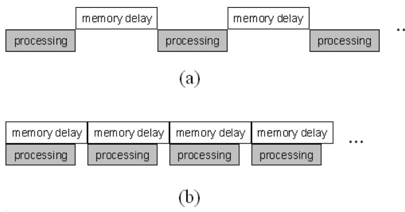

Fig. 4. SAD8 execution diagrams

Please note that all software prefetching techniques will fail to fully schedule the complete loop access pattern, since the stride is continuously adjusted from call to call and not known at compile time. Although this loop can be optimized by applying specific SIMD instructions (for ISA extensions like MMX, AltiVec and 3DNow!), the memory bandwidth requirements will remain unchanged.

The solution: SAD Flux Cache. Taking into account that the stride is usually much bigger than the cache line size, the SAD8 and SAD16 execution is envisioned to involve many stall cycles due to the main memory latency as indicated on Figure 4(a).

In such case the processor is supposed to wait for the main memory to provide the requested data that is often not resident in the cache. Please note that by processing we mean the execution of all instructions involved in a single loop iteration. The reference to the *ptr_cur*[0] and *ptr_ref*[0] will bring all of the elements needed (and maybe more data) into the cache line (indicated as "memory delay" in our figure).

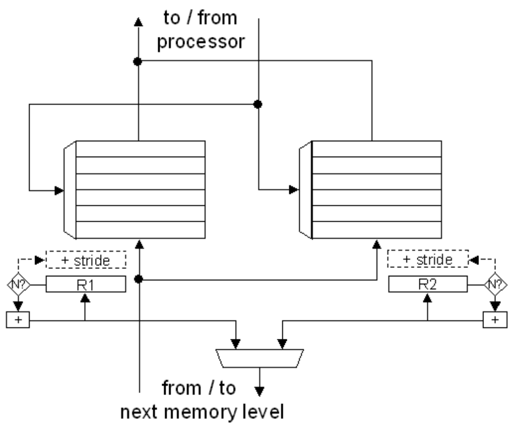

Fig. 5. SAD8 prefetch flux cache

The optimal case will be to have a prefetch strategy in hardware (prefetch engine) that mimics the memory access patterns of both SAD8 and SAD16 kernels. Considering the fact that the start addresses and the stride are changed dynamically, it should be possible to pass this information to the prefetch engine on run-time (every time the procedure call is initiated). This ideally should be done without any additional burden for the processor ISA.

It should be noted that all of the above can be done fairly easy by applying a specialized flux cache (a very small cache installed/deinstalled on demand). The proposed organization is shown in Figure 5. It consists of two stream buffers that are filled from the main memory locations indicated by the values stored in $R1$ and $R2$. The flux cache control is not only responsible for incrementing the two pointers but will also check for the loop boundaries and apply the stride offset when necessary. We apply the two stream buffers to fully utilize the available main memory bandwidth by exploiting properties like interleaving. Since the prefetching is completely decoupled from the program execution and there are two "channels" applied, the memory accesses can be performed in a back to back fashion as shown in Figure 4(b).

```
uint32_t sad8_c(const uint8_t * const cur,
       const uint8_t * const ref,
       const uint32_t stride)
{
    uint32_t sad = 0;
    uint32_t j;
    uint8_t const *ptr_cur = cur;
    uint8_t const *ptr_ref = ref;
    __asm("movtx xr1,cur");
    __asm("movtx xr2,ref");
    __asm("movtx xr3,stride");
    __asm("movtx xr4,#8");
    __asm("set $SAD_prefetch_flux");
    for (j = 0; j < 8; j++) {
        sad += abs(ptr_cur[0] - ptr_ref[0]);
        ... ... ... ... ... ... ... ... ...
        sad += abs(ptr_cur[7] - ptr_ref[7]);
        ptr_cur += stride;
        ptr_ref += stride;
    }
    __asm("set $2k_16_DM_LRU");
    return sad;
}
```

Fig. 6. Modified SAD8 C code

This is possible since both data addresses are known at advance (and are usually far away from each other), so the location of each memory read can be perfectly predicted and pre-scheduled. This all results in a highly effective prefetch strategy that in addition has a limited hardware cost.

The proposed flux cache will be installed before the execution of the SAD loop and its interface works as follows. The $R1$ and $R2$ are the two address pointers to the current (cur) and the reference (ref) arrays. These pointers are passed to the hardware prefetch engine together with the stride ($stride$) and the loop length ($N = 8$ or 16) parameters on subroutine call boundary. We envision an implementation of the proposed engine in the MOLEN polymorphic processor scenario: flux cache plus exchange registers bank for parameters passing. Please note that this is a slightly more complicated MOLEN utilization than the one described in [1]. The stream buffers size is limited and envisioned to be no more than two loop iterations, e.g. 32 entries in case of SAD16. The required buffer length can be estimated from the ratio of the average memory latency and the expected loop execution time (both measured in processor cycles). In addition, very limited control logic for scheduling of the fetches from the main memory is required. The hardware complexity of such control logic is in the order of four binary counters and one multiplexor. Please note that any specific memory burst mode can be implemented into the prefetch controller to exploit the particular memory bandwidth and resources (e.g. DMA controllers) available in the specific targeted system. The latter does not necessarily increase the complexity of the proposed control hardware.

The modified SAD8 loop for the MOLEN programming paradigm [18] is shown in Figure 6. The four additional *movtx* MOLEN instructions at the beginning are to "instruct" the SAD prefetch engine about the array access pattern and the loop boundaries as described earlier. We added the inline assembly code by hand, however the generation and the scheduling process can be integrated in the MOLEN compiler [19]. Please note the mapping of the flux cache *put* onto the MOLEN *set* instructions as previously proposed in [1].

As indicated earlier the same flux cache can be used for both SAD8 and 16 without any future modifications. The selection between the two kernels is done by the constant stored in $xr4$ (8 or 16). After loop completion the cache configuration used for the "overall" MPEG4 application code is to be restored by the second *set* instruction. Please note that the reconfiguration latency is not a major point of concern. Assuming sufficient hardware resources are available, and considering the limited size of the proposed SAD flux cache, both flux caches (SAD_prefetch_flux and 2k_16_DM_LRU) can be resident in the configuration memory (the reconfigurable hardware) at the same time. This will reduce the configuration overhead of the two *set* operations down to a trivial multiplexer switch of the address and the data busses. In addition, such scenario will prevent the cold start effects for the flux cache used for the "overall" code. For the SAD flux cache the cold start is a minor concern, since such a behavior is inherent to its functionality, e.g. the pointers and the stride are reused in very rare situations among two subsequent calls. The only overhead of the proposed flux cache will remain the four additional register to register transfer *movtx* instructions that have no impact on the main memory bus utilization.

4 Results

The improvements of the cache miss ratios of the proposed design are shown in Figure 7. The four bars (from left to right) represent the following cases: no flux cache (the base for the comparison), flux cache for SAD16 only, SAD8 flux cache and general SAD (8 and 16) flux cache as proposed in Section 3. The "remainder" of the code is using a similar 2k/16 direct mapped data cache as in the reference case (complete application without flux cache). It is interesting to note that 2k

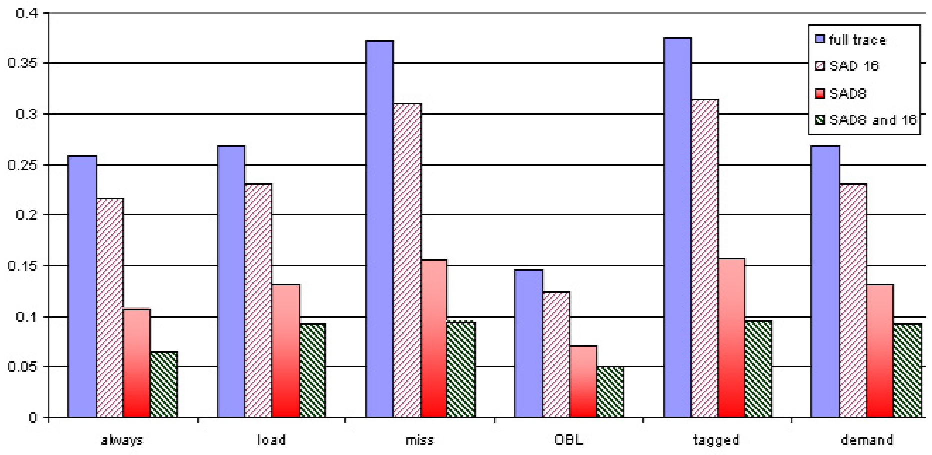

Fig. 7. MPEG4 encoder 2K DM cache and flux cache for the kernels

direct mapped cache without prefetch in combination with a SAD flux cache for the SAD8 kernel only performs better (cache miss ratio of 13%) than a 2k DM cache with OBL prefetch strategy (14.5%) that is the best performing standard prefetch mechanism for the complete application. In addition, when our flux cache is applied to both kernels the miss ratios are reduced down to the range 4.5% (for OBL) - 9.5% (for tagged prefetch). That is an improvement by near 3x. When our flux cache is applied the number of prefetches (and their impact on the main memory bus) will be reduced by a number close to the cumulative SAD8 and SAD16 memory read accesses. This forms one additional advantage of our proposal especially in the envisioned constrained embedded system context.

5 Conclusions

In this paper, we investigated the data memory access behavior of xvid MPEG4 encoder, identified kernels that can benefit form a dedicated prefetch mechanism and proposed a flux cache design to cope with it. More precisely, we studied the memory patterns of SAD8 and SAD16 during the MPEG4 encoding process. We proposed a flux cache design that optimally utilizes the main memory bandwidth with a trivial hardware complexity. We showed that our approach can reduce the data cache miss ratios by a factor close to 3x and expect to significantly reduce the number of main memory accesses when the proposed prefetching is applied. Our study focused on rather small data cache sizes (the case for very small, power constrained embedded systems) however similar relative improvements are envisioned for any realistically chosen configuration.

References

1. Gaydadjiev, G.N., Vassiliadis, S.: Flux caches: What are they and are they useful? In: Proceedings of the 5th International Workshop on Computer Systems: Architectures, Modelling, and Simulation (SAMOS 2005). (2005) 93–102
2. VanderWiel, S.P., Lilja, D.J.: Data prefetch mechanisms. ACM Computing Surveys **32** (2000) 174–199
3. Smith, A.J.: Sequential program prefetching in memory hierarchies. IEEE Computer 11 **12** (1978) 7–21
4. Vassiliadis, S., Wong, S., Gaydadjiev, G.N., Bertels, K., Kuzmanov, G., Panainte, E.M.: The molen polymorphic processor. IEEE Transactions on Computers (2004) 1363– 1375
5. Lin, W.F., Reinhardt, S.K., Burger, D.: Reducing DRAM latencies with an integrated memory hierarchy design. In: HPCA. (2001) 301–312
6. Gornish, E.H., Veidenbaum, A.: An integrated hardware/software data prefetching scheme for shared-memory multiprocessors. Int. J. Parallel Program. **27** (1999) 35–70
7. Chen, T.F.: An effective programmable prefetch engine for on-chip caches. In: MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture, Los Alamitos, CA, USA, IEEE Computer Society Press (1995) 237–242

8. Zhang, Z., Torrellas, J.: Speeding up irregular applications in shared-memory multiprocessors: memory binding and group prefetching. In: ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture, New York, NY, USA, ACM Press (1995) 188–199
9. Wang, Z., Burger, D., McKinley, K.S., Reinhardt, S.K., Weems, C.C.: Guided region prefetching: a cooperative hardware/software approach. In: ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture, New York, NY, USA, ACM Press (2003) 388–398
10. Corbal, J., Espasa, R., Valero, M.: Three-dimensional memory vectorization for high bandwidth media memory systems. In: MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture, Los Alamitos, CA, USA, IEEE Computer Society Press (2002) 149–160
11. Kuzmanov, G., Gaydadjiev, G.N., Vassiliadis, S.: Visual data rectangular memory. In: Proceedings of the 10th International Euro-Par Conference (Euro-Par 2004). (2004) 760–767
12. Edler, J., Hill, M.D.: Dinero IV trace-driven uniprocessor cache simulator. (1998) http://www.cs.wisc.edu/˜markhill/DineroIV.
13. Smith, A.J.: Cache Memories. Computing Surveys **14** (1982) 473–530
14. Burger, D., Austin, T.M., Bennett, S.: Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308 (1996)
15. (`http://ise.stanford.edu/labsite/ise_test_images_videos.html`)
16. (`http://www.itu.int/rec/recommendation.asp?lang=en\&parent=T-REC-H.261`)
17. (`http://ffmpeg.sourceforge.net`)
18. Vassiliadis, S., Gaydadjiev, G.N., Bertels, K., Panainte, E.M.: The molen programming paradigm. In: Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation. (2003) 1–10
19. Panainte, E.M., Bertels, K., Vassiliadis, S.: Compiling for the molen programming paradigm. In: Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL'03). (2003) 900–910