Flux Caches: What Are They and Are They Useful?

Georgi N. Gaydadjiev and Stamatis Vassiliadis

Computer Engineering, EEMCS, TU Delft, The Netherlands
{G.N.Gaydadjiev, S.Vassiliadis}@ewi.tudelft.nl
http://ce.et.tudelft.nl

Abstract. In this paper, we introduce the concept of flux caches envisioned to improve processor performance by dynamically changing the cache organization and implementation. Contrary to the traditional approaches, processors designed with flux caches instead of assuming a hardwired cache organization change their cache "design" on program demand. Consequently program (data and instruction) dynamic behavior determines the cache hardware design. Experimental results to confirm the flux caches potential are also presented.

1 Introduction

To improve processor performance numerous cache organizations have been proposed (and some of them implemented) in the past. All well known cache organizations can be divided in two classes: A) static approaches, e.g. victim [1, 2] [1], column associative [4], skewed-associative [5] and assist [6] caches; and B) adaptive designs, e.g split temporal/spatial [7], dual data [8], reconfigurable [9] and configurable line size [10] caches. The first group relies on time invariant design improvements, while the second one aims on trivial cache organization changes according to some running application requirements. We envision a third approach, termed *flux caches*, based on demand driven cache designs and implemented using for example reconfigurable technologies.
Reconfigurable hardware extensions of general purpose processors (GPP) have been mainly focusing on accelerating frequently used code in hardware [11, 12, 13]. Such hardware/software repartitioning usually leads to drastic changes in cache behavior since the application temporal and spacial locality is mainly accounted on highly iterative loops that form the primary subjects for hardware implementation. While dealing with the aforementioned effects did not get unnoticed [14], using on-demand hardware designs to improve the GPP memory sub-system seems to lack attention from the research community. In this paper depending on expected execution benefits of a single program (or a subsection of a program), memory sub-system designs are changed on demand. If during program execution (or before the execution of a program) it is found or expected that a different cache organization is beneficial then a new cache design is (dynamically) installed in hardware. In essence our approach allows on demand L1, L2 cache designs where all cache parameters (e.g. associativity, total size, line size,

[1] This is the earliest work on victim caches presented before the widely recognized victim cache paper of Jouppi [3].

T.D. Hämäläinen et al. (Eds.): SAMOS 2005, LNCS 3553, pp. 93–102, 2005.

replacement policy, victim cache addition etc.) can be adjusted. Using reconfigurable technologies we show how to incorporate our approach with no need for architectural changes. We target an existing processor platform [15] and show that dynamic cache design can be transparently done with no architectural (ISA) changes.

The remainder of this paper is organized as follows. Section 2 introduces the flux caches and how they map to the MOLEN machine organization. Section 3 reviews the most relevant related work. In Section 4 the simulation framework for this study and the performance results are described. Finally, the discussion is concluded in Section 5.

2 Flux Caches Organization and Implementation

It is envisioned that different programs have unique cache requirements that can be satisfied by alternative cache organizations. Support for such flexibility is expected to exploit significant improvements in application execution times. For example, let us consider two applications of different kind running on the same embedded (e.g. in a mobile phone) processor. The first one is a digital video processing algorithm with predominantly streaming (spacial locality) memory accesses. Let the second application be a Java Virtual Machine (JVM) with heavy temporal locality memory accesses. Obviously the system designer is confronted with a dilemma considering the fact that drowsy behavior for both cases is considered unacceptable. Coming up with a cache design that works optimally for both applications is rather difficult. Let assume instead that both applications can at advance (before they start) set up a cache design that will best fit their particular memory requirements. This is not such a non-realistic scenario since in the majority of the cases the user will never watch a football match and play a strategy game at the same time. In such a system, different cache designs coexist in time with their corresponding applications and can be optimized according to the specific demands.

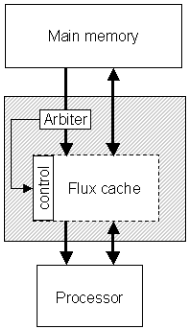


Fig. 1. Flux cache

Flux caches are fully customizable memories, possibly implemented in reconfigurable hardware, that can be installed on demand before or during program execution. Hardware implementations of arbitrary cache design can be instantiated under software or hardware control at runtime and are pre-determined ”off-line” at hardware/software co-design stage using application partitioning, monitoring, profiling etc. Generally speaking the flux cache mechanism would require additional ISA support [2] to enforce the intended cache design and will introduce some reconfiguration overhead. The flux cache organization is depicted in Figure 1.

The *Arbiter* will partially decode the instructions received from the instruction fetch unit and issue the flux cache instructions to the *control* unit. The control unit is responsible for loading the cache configuration code from memory and instruction / data paths consistency. The envisioned operations support consists only of a single *put*

[2] But not always as it will be shown later by using the MOLEN polymorphic processor [15].

phase. During this phase, the flux cache is configured to the intended hardware organization. More precisely, a bitstream is loaded from the main memory into the local configuration memory. This concept requires one-time architectural extension by a single instruction. The **put** instruction that initiates the flux cache configuration has the following format: *put* <*address*>. The *address* is a memory location the first element of the configuration bitstream is to be loaded from. Parameters of the cache are usually implicit as in the example presented hereafter, explicit calls can also be envisioned. The *put* phase is initiated by the arbiter after detection of a **put** instruction and has to be interrupted right after the hardware configuration is completed. This can be achieved only by proper configuration bitstream termination. There exist two different approaches (using special operation at the end or by defining the configuration code length at the beginning) both with their advantages and shortcomings.

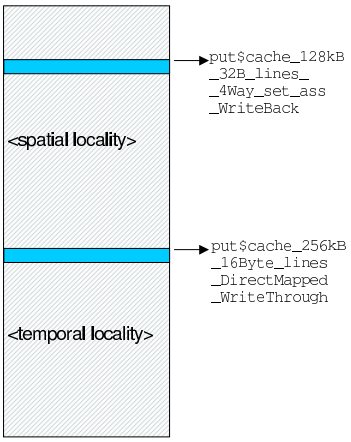


Fig. 2. Single program execution

Assuming the case (different from the aforementioned two applications example) of a single application with clearly defined regions with predominant spacial or temporally localities executing on a machine augmented with a flux cache (Figure 2). The original GPP execution code sequence is augmented with **put** instructions at the positions different cache organization is needed. The decision on cache type, size and configuration is left to the system designer since he/she is expected to understand the targeted applications behavior. The cache selection process can be supported by profiling, cache simulation and/or dynamic program monitoring. In addition, the latter process can be fully automated and integrated in the automated design tools. The **put** instruction will be redirected to the control unit and interpreted. 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 GPP will continue from the next instruction following the **put**. In order to reduce the penalty of such execution stalls various prefetch and partial configuration techniques [16] and concurrent loading can be applied. Please note that after complete reconfiguration, the ”new” cache will be ”empty” and the cold-start effects have to be taken into consideration (keeping ”old” filled caches, prefetching designs to fill caches and partial flux cache designs may help). The flux caches can be realized using existing technology, i.e. Virtex II Pro platform FPGA from Xilinx. The only constraint on the targeted technology is partial reconfiguration support.

To show the flux cache feasibility we assume reconfigurable implementation and the MOLEN paradigm. The MOLEN machine organization consists of two main com-

ponents: the Core Processor (CP), usually a general purpose processor, and the Reconfigurable Processor (RP).

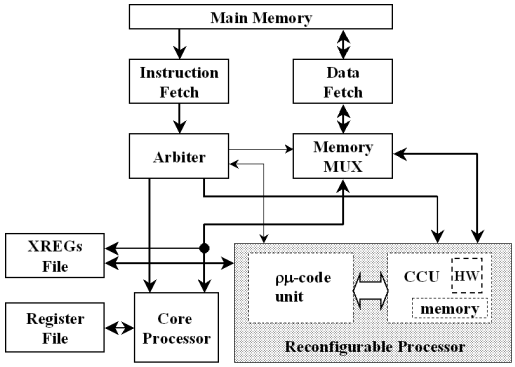


Fig. 3. MOLEN organization

The application's division in a hardware and a software part is directly mappable to the above two units. The execution flow redirection is performed by the Arbiter using partial instruction decoding. In respect to the Core Processor original ISA, MOLEN requires only an one-time extension with four and up to eight instructions dependent on the specific implementation [16]. To perform the actual reconfiguration of the CCU, reconfiguration microcode is loaded into the ρμ-code unit. This stage is also referred to as the **set** phase. The **execute** phase is responsible for the actual operation execution on the CCU, and is performed by running the *execution microcode*. It is important to emphasize that both the **set** and **execute** phases do not specify any pre-defined hardware operation to be performed. Instead, the **pset**, **cset** and **execute** instructions (*reconfigurable instructions*) directly point to the memory location where the reconfiguration or execution microcode is located. The hardware/software communication is supported by the Exchange Registers bank and performed through the **movtx** and **movfx** MOLEN instructions. As depicted on Figure 4, flux caches can be implemented under a simplified MOLEN scenario (only flux caches no CCUs). Cache coherence logic may be needed if for example the core processor employs L1 caches. The *put* flux cache phase is functionally equivalent to the *set* phase in MOLEN. All MOLEN configuration microcode termination and prefetching techniques [13] are directly applicable to flux caches. Said this we can use the MOLEN **set** instruction for **put** emulation. The execution phase with its supporting MOLEN instructions and functional modules is no longer needed for the flux cache implementation case. This allows the overal system organization to be reduced significantly. First of

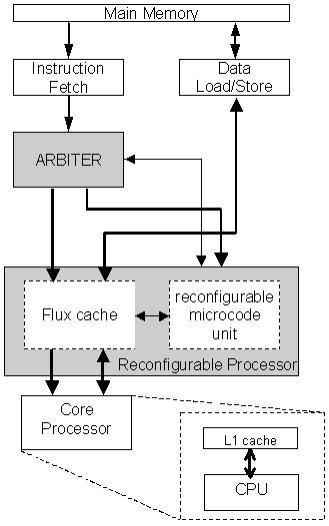


Fig. 4. A flux cache implementation

all the data memory Multiplexer / Demultiplexer can be avoided due to the absence of CCU that will perform data accesses. The exchange registers bank and the two *move* instructions for data exchange between the core and the reconfigurable processors are not required. The sequential consistency model, inherent to MOLEN, will naturally arbitrate the execution of the core processor code with the flux cache reconfiguration times. This leads to a very minimal but still completely functional flux cache implementation that will decrease the complexity (and the overall overhead) of the MOLEN functional blocks. In reality, the arbiter and the simplified $\rho\mu$-code unit can be combined into a single module that handles the flux cache reconfigurations. It is to be noted that code running on such simplified MOLEN instantiation will be binary compatible with any other MOLEN implementation. In the opposite direction, however, additional fix up code and exception handling may be needed to cope with all not implemented MOLEN (e.g. *mov* and *execute*) instructions.

3 Related Work

The flux caches allow their internal structure to be ”redesigned” at any given moment during the execution time. This is the reason why only the time variant cache proposals (as introduced in Section 1) will be considered hereafter.

The ”reconfigurable caches” introduced by Ranganthan et. al. [9] divide the available cache memory into several partitions that may be used to support applications usually unable to exploit conventional caches in an optimal way. As example the multimedia applications with their streaming nature are used. Although named reconfigurable, this proposal is just an extension of the conventional set-associative and direct mapped cache designs to support a limited number of partitions that are dynamically selectable. In addition, special ISA support may be required to control repartitioning (in case the software controlled approach is used). Our proposal differs in two aspects: first we do not impose any limitation on the number of possible cache configurations; and second very limited or no additional ISA support (as in the case of the MOLEN processor) is required to indicate the intended configuration.

The Split Temporal/Spatial (STS) caches [7] employ two cache sub-systems: one for ”temporal” data and another for ”spacial” data. The main idea is that handling data with temporal locality in a ”spacial” way, e.g. prefetching its neighboring addresses is usually counterproductive. This leads to data classification into two sub-groups, each to be handled separately by the corresponding cache. Such classification can be performed on compile / profile or run-time. Two ways to express this to the hardware are envisioned: by ISA extension or by tagging. The flux caches differ from STS caches in the following way. First, we allow instruction and data cache modifications compared to data cache only target of the STS caches. Second, we do not require and additional ISA modifications or tag bits to implement similar functionality. STS caches can be implemented in flux caches in a straight forward way by using the MOLEN *pset* or *execute* instructions to distinguish between ”temporal” and ”spacial” data.

The Dual Data Cache (DDC) bears some similarities with STS. Like STS, it has two separate modules to deal with data of different locality. The data allocation, however, is more sophisticated and one additional *bypass* mode is introduced. The memory

instructions are tagged as in STS with the difference that five different data types are distinguished. As in the case of STS caches our proposal differs in its flexibility concerning the instruction cache and its zero overhead ISA support.

The configurable line size caches proposed by the University of California, Riverside [10] focus mainly on the cache memory energy consumption. This work covers static selection of the cache line size early in the embedded system design process. The assumption is that an embedded system will execute only a pre-defined (and hence very limited) set of applications during its operational lifetime. Later ongoing research of the same group [14] reported dynamic configuration during run-time. However, again only a very limited number of cache configurations is supported. Our proposal does not impose such restriction on the system designers, allowing them to introduce changes later (even in the field) when new applications have been added or existing one should be upgraded (e.g. using MPEG4 instead of MPEG2). The above list of related work is not complete but to our knowledge representative. The reason of not including all previous approaches is the significant number of publications on the topic and space limitations. Proposals such as software managed data caches (implemented in HP PA-7200 CPU) [17] or Veidenbaum's et. al. dynamic cache line size adaptation [18] are not considered in details due to some similarities with DDC and the work from UC Riverside respectively.

All of the proposals reported in the publicly available literature do focus on organizing the available cache memory in a number of pre defined ways, mainly in respect to associativity and cache line size. In our proposal the only restriction known is the available reconfigurable hardware resources (e.g. on-chip SRAM size) that may limit the overal cache size. All remaining cache parameters, e.g. replacement strategy, prefetching and write back policy, can be adjusted to the targeted application in order to gain optimal performance. In addition, our proposal does not limit the system designer to the conventional cache architectures and provides him with means to utilize (and/or evaluate) unique approaches, e.g. stream caches, or even design and apply completely customized memory sub-system (e.g. 2-D rectangular memory [19]).

4 Simulation Framework, Methodology and Results

We studied the potential benefits of reconfigurable caches using dinero IV [20], a trace driven cache simulator that models the first two levels of the memory hierarchy. We share the opinion that statements about cache performance can be based only on trace-driven simulation or direct measurements [21]. The former method is slow and has significant demands on storage capacity, while the latter is fast but prohibitively expensive. Since our study is about relative cache performance, a non-functional simulator such as dinero is considered sufficient. The application traces for this study where obtained using the SimpleScalar 4.0 simulator [22] modified to generate dinero style memory traces. The traces where generated in an in-order execution fashion. Only the three basic memory access types where implemented: data read, data write and instruction fetch. This fact, however, does not have any influence on the generality of the reported results. The targeted applications of interest where multimedia.

Table 1. Benchmarks used in this study

benchmark	Description	Input
gsmenc	GSM speech encoding (toast)	clinton.pcm
gsmdec	GSM speech decoding (untoast)	clinton.pcm.gsm
adpcmenc	ADPCM speech encoding (rawcaudio)	clinton.pcm
adpcmdec	ADPCM speech decoding (rawdaudio)	clinton.adpcm
mpegenc	MPEG-2 video encoding (four 352x240 frames IBBP)	mei16v2.yuv
mpegdec	MPEG-2 video decoding (video stream to YUV)	mei16v2.m2v
cjpeg	JPEG encoding (1024x630 3-band image)	rose16.ppm
djpeg	JPEG decoding (1024x630 3-band image)	rose16.jpg
epicenc	EPIC encoding (unepic) (512x512 grayscale image)	test.image.pgm.E
epicdec	EPIC decoding (epic) (512x512 grayscale image)	test_image.pgm
g721enc	G721 speech compression	clinton.pcm
g721dec	G721 speech decompression	clinton.g721

This is the reason for selecting a representative set of benchmarks and corresponding data sets from the UCLA MediaBench [23] suite as summarized in Table 1. We targeted set of benchmarks that cover audio, video, images and speech data processing that is assumed to represent the application domain for our study. We have simulated many different L1 caches to explore the impact of various cache parameters on the miss ratio. All the simulation and data collection work was automated using a script that did attempt local and global minimum determination in the reported miss ratios. Sophisticated algorithms for optimal cache selection are outside the scope of the current study.

We do realize that some of the synthetic benchmarks used may not truly represent a real-life multimedia application. For example, the gsm pair (also known as toast/untoast) consists mainly of highly iterative functions that rely on the *register* keyword for speed up optimizations. In our case (SimpleScalar architecture and gcc compiler) unrealistically low data miss ratios are expected for those benchmarks. On the other hand, such situation forms a worst case scenario for evaluation of the proposed cache organization.

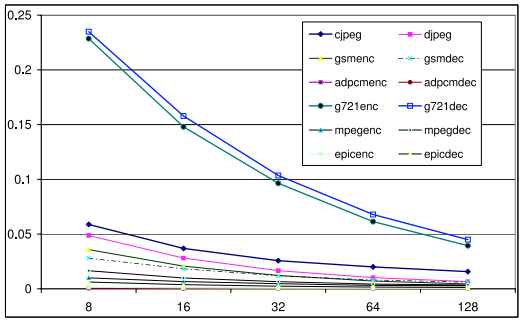
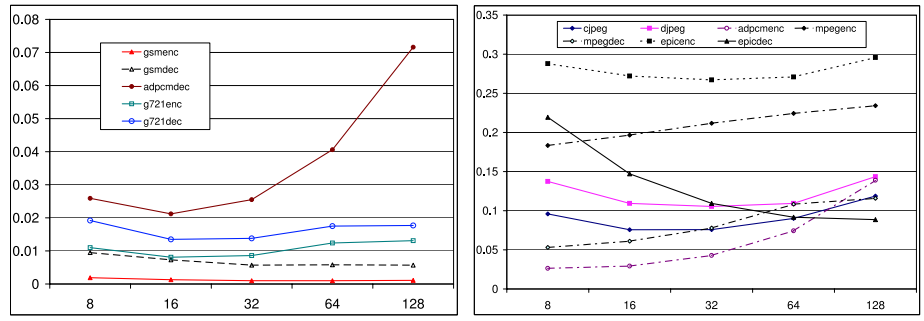


Fig. 5. I-cache miss ratios vs line size

In an attempt to evaluate the optimal configuration for the targeted benchmark set under a flux cache scenario, a variety of cache configurations where simulated. They all differ in overall cache sizes, line sizes, associativity, prefetch behavior and write-allocate and write-back policies just to name a few. For simplicity, we always assumed



(a) (b)

Fig. 6. D-cache miss ratios vs line size

only L1 split instruction/data caches of equal sizes. The primary cache size of interest is 8k (2x4k instruction and data caches) - a realistic scenario for the embedded domain. We did not evaluate the influence of the replacement policy since it has been found that LRU and FIFO outperform the random approach, however do not show significant differences among each other. In all of the experiments reported hereafter the LRU replacement is used. We would like to emphasize that this is only due to the specific behavior of the targeted benchmarks and does not form any restriction for the proposed approach. It is very likely that different applications may greatly benefit from replacement policy changes. The first well expected clear difference in performance was found when the cache line size was changed.

Figures 5 and 6 depict how the cache line size influences the miss ratio. The instruction cache miss ratio is shown in Figure 5, while Figure 6 (a) and (b) demonstrate the miss ratio variation for the data cache. In this experiment, direct mapped cache with instruction and data cache sizes equal to 4k where considered. While for the instruction cache the miss ratio keeps decreasing with increasingly larger cache line sizes, the data cache miss ratio shows a clear optimum at certain sizes. For example the *djpeg* and *cjpeg* curves have a minimum at 16 and 32 byte line sizes. The *adpcm* encoder and decoder perform optimally with 8 and respectively 16 bytes long cache lines. Two benchmarks show slightly deviating behavior - the *epic* and the *mpeg*. Both *mpeg* variants, the encode and the decode, show increasing miss ratios when the line size grows from 8, through, 16, 32, 64 and up to 128 bytes. *Epic* however shows even more surprising properties - while the encode direction shows clear miss ratio minimum in the miss ratio for cache line size of 32 bytes, the decoding part of the benchmark shows a minimum only at 128 byte cache line. This fact, however not shown in the figure was found by performing experiments with 256 byte cache lines. The remaining two benchmarks-*gsm* and *g721* do not show significant changes in our experiment, mainly due to the usage of the C *register* keyword that will assign most of the variables to internal registers. It is interesting to note, however that the instruction cache behavior for the *g721* encoder and decoder shows heavy dependence on the cache line size. To summarize, the optimal flux cache configuration needed for *djpeg* and *cjpeg* should be 4k/32 (4k cache organized into 32 byte lines) for instructions and 4k/32 for data for optimal cache

performance. Please note that we did ignore some minor differences in cache miss ratios for 32 byte (0.0257) and 128 byte (0.0157) cases otherwise we would be selecting 4k/128 configuration for the *cjpeg* instruction cache. The same configuration (4k/32) works best for the *epic* encoder, while before starting the *epic* decoder the flux cache is to be ”redesigned” into a 4k/128 considering optimal data cache performance.

5 Conclusions and Future Work

In this paper, we introduced the concept of flux caches and have indicated their performance potential for applications with streaming data access patterns such as multimedia. More precisely, we studied different cache sizes and showed the improvement potential inherent to the studied applications in respect to the line size in the case of 8k cache. Since cache miss ratios do only give an indication about the flux cache performance, currently we are implementing the flux caches on the MOLEN Virtex-II Pro prototype and will report the measured numbers in the near future. In addition, the energy performance analysis of the proposed organization needs careful investigation, together with issues like: data consistency and multiprogramming environment.

References

1. Dejuan, E., Casals, O., Labarta, J.: Cache memory with hybrid mapping. In: 7th International Conference on Modelling, Identification and Control, Grindelwald (1987) 27–30
2. Dejuan, E., Casals, O., Labarta, J.: Management algorithms for an hybrid mapping cache memory. In: International Conference on Mini an Microcomputers and their applications, Sant Feliu (1988) 368–372
3. Jouppi, N.P.: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In: ISCA. (1990) 364–373
4. Agarwal, A., Pudar, S.D.: Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. In: ISCA. (1993) 179–190
5. Seznec, A.: A case for two-way skewed-associative caches. In: ISCA. (1993) 169–178
6. Chan, K.K., Hay, C.C., Keller, J.R., Kurpanek, G.P., Schumacher, F.X., Zheng, J.: Design of the HP PA 7200 CPU processor chip. Hewlett-Packard Journal **47** (1996) 25–33
7. Milutinovic, V., Markovic, B., Tomasevic, M., Tremblay, M.: The split temporal/spatial cache: Initial performance analysis. Proceedings of SCIzzL-5 (1996) 63–69
8. Sánchez, F.J., González, A., Valero, M.: Software management of selective and dual data caches. In: Technical Committee on Computer Architecture (TCCA) Newsletter. (1997)
9. Ranganathan, P., Adve, S.V., Jouppi, N.P.: Reconfigurable caches and their application to media processing. In: ISCA. (2000) 214–224
10. Zhang, C., Vahid, F., Najjar, W.A.: Energy benefits of a configurable line size cache for embedded systems. In: ISVLSI. (2003) 87–91
11. Hartenstein, R.W., Kress, R., Reining, H.: A new FPGA Architecture for Word-Oriented Datapaths. In: 4th International Workshop on Field Programmable Logic and Applications:Architectures, Synthesis and Applications. (1994) 144–155
12. Trimberger, S.M.: Reprogramable Instruction Set Accelerator. U.S. Patent No. 5,737,631 (1998)

13. Vassiliadis, S., Wong, S., Cotofana, S.: The MOLEN ρμ-Coded Processor. In: 11th International Conference on Field Programmable Logic and Applications (FPL). Volume 2147., Belfast, UK, Springer-Verlag Lecture Notes in Computer Science (LNCS) (2001) 275–285
14. Gordon-Ross, A., Vahid, F., Dutt, N.: Automatic tuning of two-level caches to embedded applications. In: DATE. (2004) 208–213
15. 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
16. 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
17. Kurpanek, G., Chan, K., Zheng, J., DeLano, E., Bryg, W.: Pa7200: A pa-risc processor with integrated high performance mp bus interface. In: COMPCON. (1994) 375–382
18. Veidenbaum, A.V., Tang, W., Gupta, R., Nicolau, A., Ji, X.: Adapting cache line size to application behavior. In: ICS '99: Proceedings of the 13th international conference on Supercomputing, New York, NY, USA, ACM Press (1999) 145–154
19. 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
20. Edler, J., Hill, M.D.: Dinero IV trace-driven uniprocessor cache simulator. (1998) http://www.cs.wisc.edu/~markhill/DineroIV.
21. Smith, A.: Cache Memories. Computing Surveys **14** (1982) 473–530
22. Burger, D., Austin, T.M., Bennett, S.: Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308 (1996)
23. Lee, C., Potkonjak, M., Mangione-Smith, W.H.: Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In: 30th Annual International Symposium on Microarchitecture, MICRO30. (1997) 330–335