Memory Access Schemes for Configurable Processors

Holger Lange and Andreas Koch

Tech. Univ. Braunschweig (E.I.S.), Gaußstr. 11, D-38106 Braunschweig, Germany
lange,koch@eis.cs.tu-bs.de

Abstract. This work discusses the Memory Architecture for Reconfigurable Computers (MARC), a scalable, device-independent memory interface that supports both irregular (via configurable caches) and regular accesses (via pre-fetching stream buffers). By hiding specifics behind a consistent abstract interface, it is suitable as a target environment for automatic hardware compilation.

1 Introduction

Reconfigurable compute elements can achieve considerable performance gains over standard CPUs [1] [2] [3] [4]. In practice, these configurable elements are often combined with a conventional processor, which provides the control and I/O services that are implemented more efficiently in fixed logic. Recent single-chip architectures following this approach include NAPA [5], GARP [6], OneChip [7], OneChip98 [8], Triscend E5 [9], and Altera Excalibur [10]. Board-level configurable processors either include a dedicated CPU [11] [12] or rely on the host CPU for support [13] [14].

Design tools targeting one of these hybrid systems such as GarpCC [15], Nimble [16] or Napa-C [17] have to deal with software and hardware issues separately as well as with the creation of interfaces between these parts. On the software side, basic services such as I/O and memory management are often provided by an operating system of some kind. This can range from a full-scale general-purpose OS over more specialized real-time embedded OSes down to tiny kernels offering only a limited set of functions tailored to a very specific class of applications. Usually, a suitable OS is either readily available on the target platform, or can be ported to it with relative ease.

This level of support is unfortunately not present on the hardware side of the hybrid computer. Since no standard environment is available for even the most primitive tasks such as efficient memory access or communication with the host, the research and development of new design tools often requires considerable effort to provide a reliable environment into which the newly-created hardware can be embedded. This environment is sometimes called a *wrapper* around the custom datapath. It goes beyond a simple assignment of chip pads to memory pins. Instead, a structure of on-chip busses and access protocols to various resources (e.g., memory, the conventional processor, etc) must be defined and implemented.

In this paper, we present our work on the Memory Architecture for Reconfigurable Computers (MARC). It can act as a "hardware target" for a variety of hybrid compilers, analogously to a software target for conventional compilers. Before describing its specifics, we will justify our design decisions by giving a brief overview of current configurable architectures and showing the custom hardware architectures created by some hybrid compilers.

2 Hybrid Processors

Static and reconfigurable compute elements may be combined in many ways. The degree of integration can range from individual reconfigurable function units (e.g., OneChip [7]) to an entirely separate coprocessor attached to a peripheral bus (e.g., SPLASH [4], SPARXIL [18]).

Figure 1. Single-chip hybrid processor

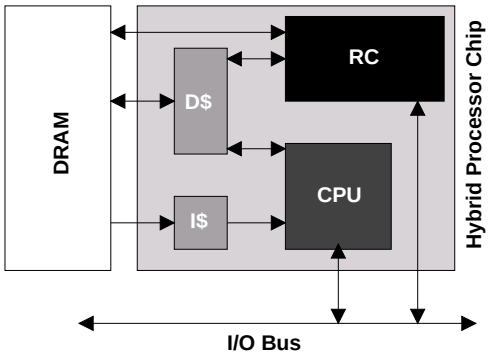
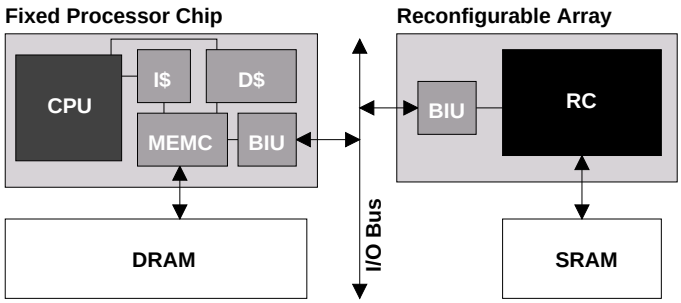


Figure 1 sketches the architecture of a single-chip hybrid processor that combines fixed (CPU) and reconfigurable (RC) compute units behind a common cache (D$). Such an architecture was proposed, e.g., for GARP [6] and NAPA [5]. It offers very high bandwidth, low latency, and cache coherency between the CPU and the RC when accessing the shared DRAM.

Figure 2. Hybrid processor emulated by multi-chip system



Operation	Cycles
ZBT SRAM read	4
ZBT SRAM write	4
PCI read	46-47
PCI write	10

Table 1. Data access latencies (single word transfers)

The board-level systems more common today use an architecture similar to Figure 2 . Here, a conventional CPU is attached by a bus interface unit (BIU) to a system-wide I/O bus (e.g., SBus [18] or PCI [11] [12]). Another BIU connects the RC to the I/O bus. Due to the high communication latencies over the I/O bus, the RC is often attached directly to a limited amount of dedicated memory (commonly a few KB to a few MB of SRAM). In some systems, the RC has access to the main DRAM by using the I/O bus as a master to contact the CPU memory controller (MEMC). With this capability, the CPU and the RC are sharing a logically homogeneous address space: Pointers in the CPU main memory can be freely exchanged between software on the CPU and hardware in the RC.

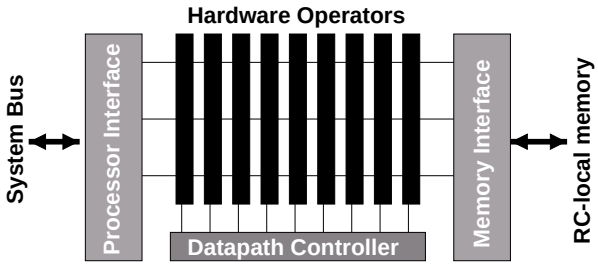
Table 1 shows the latencies measured on [12] for the RC accessing data residing in local Zero-Bus Turnaround (ZBT) SRAM (latched in the FPGA I/O blocks) and in main DRAM (via the PCI bus). In both cases, one word per cycle is transferred after the initial latency.

It is obvious from these numbers that any useful wrapper must be able to deal efficiently with access to high latency memories. This problem, colloquially known as the "memory bottleneck", has already been tackled for conventional processors using memory hierarchies (multiple cache levels) combined with techniques such as pre-fetching and streaming to improve their performance. As we will see later, these approaches are also applicable to reconfigurable systems.

3 Reconfigurable Datapaths

The structure of the compute elements implemented on the RC is defined either manually or by automatic tools. A common architecture [6] [16] [18] is shown in Figure 3 .

Figure 3. Common RC datapath architecture



The datapath is formed by a number of hardware operators, often created using module generators, which are placed in a regular fashion. While the linear placement shown in the figure is often used in practice, more complicated layouts are of course possible. All hardware operators are connected to a central datapath controller that orchestrates their execution.

In this paper, we focus on the interface blocks attaching the datapath to the rest of the system. They allow communication with the CPU and main memory using the system bus or access to the local RC RAM. The interface blocks themselves are accessed by the datapath using a structure of uni- and bidirectional busses that transfer data, addresses, and control information.

For manually implemented RC applications, the protocols used here are generally developed ad-hoc and heavily influenced by the specific hardware environment targeted. (e.g., the data sheets of the actual SRAM chips on a PCB). In practice, they may even vary between different applications running on the same hardware (e.g., usage of burst-modes, fixed access sizes etc.).

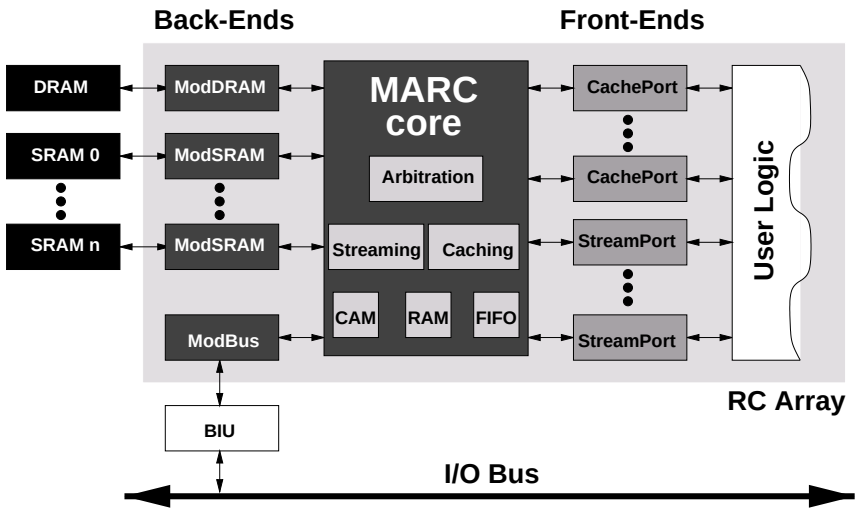
This approach is not applicable for automatic design flows: These tools require pre-defined access mechanisms to which they strictly adhere for all designs. An example for such a well-defined protocol suitable as a target for automatic compilation is employed on GARP [6], a single-chip hybrid processor architecture. It includes standardized protocols for sending and retrieving data to/from the RC using specialized CPU instructions and supported by dedicated decoding logic in silicon. Memory requests are routed over a single address and four data busses that can supply up to four words per cycle for regular (streaming) accesses.

None of these capabilities is available when using off-the-shelf silicon to implement the RC. Instead, each user is faced with implementing the required access infrastructure anew.

4 MARC

Our goal was to learn from these past experiences and develop a single, scalable, and portable memory interface scheme for reconfigurable datapaths. MARC strives to be applicable for both single-chip and board-level systems, and to hide the intricacies of different memory systems from the datapath. Figure 4 shows an overview of this architecture.

Figure 4. MARC architecture



Using MARC, the datapath accesses memory through abstract front-end interfaces. Currently, we support two front-ends specialized for different access patterns: Caching ports provide for efficient handling of irregular accesses. Streaming ports offer a non-unit stride access to regular data structures (such as matrices or images) and perform address generation automatically. In both cases, data is pre-fetched/cached to reduce the impact of high latencies (especially for transfers using the I/O bus). Both ports use stall signals to indicate delays in the data transfer (e.g., due to cache miss or stream queue refill). A byte-steering logic aligns 8- and 16-bit data on bits 7:0 and 15:0 of the data bus regardless of where the datum occurred in the 32-bit memory or bus words.

The specifics of hardware memory chips or system bus protocols are implemented in various back-end interfaces. E.g., dedicated back-ends encapsulate the mechanisms for accessing SRAM or communicating over the PCI bus using the BIU.

The MARC core is located between front- and back-ends, where it acts as the main controller and data switchboard. It performs address decoding and arbitration between transfer initiators in the datapath and transfer receivers in the individual memories and busses. Logically, it can map an arbitrary number of front-ends to an arbitrary number of back-ends. In practice, though, the number of resources managed is of course limited by the finite FPGA capacity. Furthermore, the probability of conflicts between initiators increases when they share a smaller number of back-ends. However, the behavior visible to the datapath remains identical: The heterogeneous hardware resources handled by the back-ends are mapped into a homogeneous address space and accessed by a common protocol.

4.1 Irregular cached access

Caching ports are set up to provide read data one cycle after an address has been applied, and accept one write datum/address per clock cycle. If this is not possible (e.g., a cache miss occurs), the stall signal is asserted for the affected port, stopping the initiator. When the stall signal is de-asserted, data that was "in-flight" due to a previous request will remain valid to allow the initiator to restart cleanly.

Table 2(a) describes the interface to a caching port. The architecture currently allows for 32-bit data ports, which is the size most relevant when compiling software into hybrid solutions. Should the need arise for wider words, the architecture can easily be extended.

Signal	Kind	Function
Addr	in	Address.
Data	in/out	Data item.
Width	in	8, 16, 32-bit access.
Stall	out	Asserted on cache miss.
OE	in	Output enable
WE	in	Write enable
Flush	in	Flush cache.

(a) Caching port interface

Sig/Reg	Kind	Function
Addr	reg	Start address.
Stride	reg	Stride (increment).
Width	reg	8,16,32-bit access.
Block	reg	FIFO size.
Count	reg	Length, transfer.
R/W	reg	Read or write.
Data	i/o	Data item.
Stall	out	Wait, FIFO flush/refill.
Hold	in	Pause data flow.
EOS	out	End of stream reached.
Load	in	Accept new parameters.

(b) Streaming port interface

Table 2. Port interfaces

Arbitrary memory ranges (e.g., memory-mapped I/O registers) can be marked as non-cacheable. Accesses to these regions will then bypass the cache. Furthermore, since all of the cache machinery is implemented in configurable logic, cache port characteristics such as number of cache lines and cache line length can be adapted to the needs of the application. As discussed in [19], this can result in a 3% to 10% speed-up over using a single cache configuration for all applications.

4.2 Regular streamed access

Streaming ports transfer a number of data words from or to a memory area without the need for the datapath to generate addresses. After setting the parameters of the transfer (by switching the port into a "load parameter" mode), the port presents/accepts one data item per clock cycle until it has to refill or flush its internal FIFOs. In that case, the stall signal stops the initiator using the port. When the FIFO becomes ready again, the stall signal is de-asserted and the transfer continues. The datapath can pause the transfer by asserting the hold signal. As before, our current implementation calls for a 32-bit wide data bus. Table 2(b) lists the parameter registers and the port interface.

The 'Block' register plays a crucial role in matching the stream characteristics to the specific application requirements. E.g., if the application has to process a very large string (such as in DNA matching), it makes sense for the datapath to request a large block size. The longer start-up delay (for the buffer to be filled) is amortized over the long run-time of the algorithm. For smaller amounts of data (e.g., part of a matrix row for blocking matrix multiplication), it makes much more sense to pre-fetch only the precise amount of data required. [20] suggests compile-time algorithms to estimate the FIFO depth to use.

The cache is bypassed by the streaming ports in order to avoid cache pollution. However, since logic guaranteeing the consistency between caches and streams for arbitrary accesses would be very expensive to implement (especially when non-unit strided streams are used), our current design requires that accesses through the caching ports do not overlap streamed memory ranges. This restriction must be enforced by the compiler. If that is not possible, streaming ports cannot be used. As an alternative, a cache with longer cache lines (e.g., 128 bytes), might be used to limit the performance loss due to memory latency.

4.3 Multi-threading

Note that all stall or flow-control signals are generated/accepted on a per-port basis. This allows true multi-threaded hardware execution where different threads of control are assigned to different ports. MARC can accommodate more logical ports on the front-ends than actually exist physically on the back-ends. For certain applications, this can be exploited to allow the compiler to schedule a larger number of memory accesses in parallel. The MARC core will resolve any inter-port conflicts (if they occur at all, see Section 5) at run-time. The current implementation uses a round-robin policy, later versions might extend this to a priority-based scheme.

4.4 Flexibility

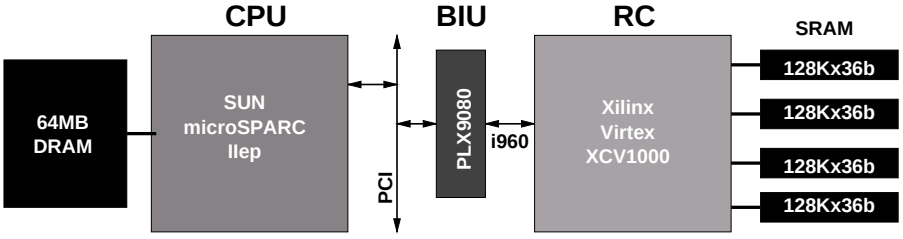
A separate back-end is used for each memory or bus resource. For example, in a system with four ZBT SRAM memories, four instances of the ZBT SRAM back-end would be instantiated. The back-ends present the same interface as a caching port (Table 2(a)) to the MARC core. They encapsulate the state and access mechanisms to manage each of the physical resources. E.g., a PCI backend might know how to access a PCI BIU and initiate a data transfer.

In this manner, additional back-ends handling more memory banks can be attached easily. Analogously, MARC can be adapted to different FPGA technologies. For example, on the Xilinx Virtex [24] series of FPGA, the L1 cache of a caching port might be implemented using the on-chip memories. On the older XC4000XL series, which has only a limited amount of on-chip storage, the cache could be implemented in a direct-mapped fashion that has the cache lines placed in external memory.

5 Implementation Issues

Our first MARC implementation is targeting the prototyping environment described in [12]. The architecture details relevant for this paper are show in Figure 5 .

Figure 5. Architecture of prototype hardware



A SUN microSPARC-IIep RISC [21] [22] is employed as conventional CPU. The RC is composed of a Xilinx Virtex XCV1000 FPGA [24].

5.1 Status

At this point, we have implemented and intensively simulated a parameterized Verilog model of the MARC Core and back-ends. On the front-end side, caching ports are already operational, while streaming ports are still under development. The design is currently only partially floor-planned, thus the given performance numbers are preliminary.

5.2 Physical Resources

The RC has four 128Kx36b banks of ZBT SRAM as dedicated memory and can access the main memory (64MB DRAM managed by the CPU) over the PCI bus. To this end, a PLX 9080 PCI Accelerator [23] is used as BIU that translates the RC bus (i960-like) into PCI and back. The MARC core will thus need PLX and ZBT SRAM back-ends. All of their instances can operate in parallel.

5.3 MARC Core

The implementation follows the architecture described in Section 4 : An arbitrary number of caching and streaming ports can be managed. In this implementation (internally relying on the Virtex memories in dual-ported mode), two cache ports are guaranteed to operate without conflicts, and three to four cache ports may operate without conflicts. If five or more cache ports are in use, a conflict will occur and be resolved by the Arbitration unit (Section 4.3). This version of the core currently supports a 24-bit address space into which the physical resources are mapped.

5.4 Configurable Cache

We currently provide three cache configurations: 128 lines of 8 words, 64 lines of 16 words, or 32 lines of 32 words. Non-cacheable areas may be configured at compile time (the required comparators are then synthesized directly into specialized logic). The datapath can explicitly request a cache flush at any time (e.g., after the end of a computation).

The cache is implemented as a fully associative L1 cache. It uses 4KB of Virtex BlockSelectRAM to hold the cache lines on-chip and implements write-back and random line replacement. The BlockSelectRAM is used in dual-port mode to allow up to two accesses to occur in parallel. Conflicts are handled by the MARC Core Arbitration logic. The CAMs needed for the associative lookup are composed from SRL16E shift registers as suggested in [25]. This allows a single-cycle read (compare and match detection) and 16 clock cycles to write a new tag into the CAM. Since this operation occurs simultaneously with the loading of the cache lines from memory, the CAM latency is completely hidden in the longer memory latencies. As this 16-cycle delay would also occur (and could not be hidden) when reading the tag, e.g., when writing back a dirty cache line, the tags are additionally stored in a conventional memory composed from RAM32x1S elements that allow single-cycle reading.

For each caching port, a dedicated CAM bank is used to allow lookups to occur in parallel. Each cache line requires 5 4-bit CAMs, thus the per-port CAM area requirements range from 160 to 640 4-LUTs. In addition to the CAMs, each cache port includes a small state-machine controlling the cache operation for different scenarios (e.g., read hit, read miss, write hit, write miss, flush, etc.). The miss penalty c_m in cycles for a clean cache line is given by

$$c_m = 7 + c_{\text{be}} + w + 4,$$

where c_{be} is the data transfer latency for the back-end used (Table 1) and w is the number of 32-bit words per cache line. 7 and 4 are the MARC Core operation startup and shutdown times in cycles, respectively. For a dirty cache line, an additional w cycles are required to write the modified data back.

For comparison with [19], note that according to [26], the performance of 4KB of fully-associative cache is equivalent to that of 8KB of direct-mapped cache.

5.5 Performance and Area

The performance and area requirements of MARC Core, the technology modules and two cache ports are shown in Table 3 .

Configuration	4LUTs	FFs	RAM32X1Ss	BlockRAMs	Clock
32x32	2844	976	12	8	31 MHz
64x16	4182	1038	26	8	30 MHz
128x8	7132	1531	56	8	29 MHz
XCV1000 avail.	24576	24576	(each uses 2 4LUTs)	32	—

Table 3. Performance and area requirements

For the three configuration choices, the area requirements vary between 10%-30% of the chip logic capacity. Since all configurations use 4KB of on-chip memory for cache line storage, 8 of the 32 512x8b BlockSelectRAMs are required.

5.6 Scalability and Extensibility

As shown in Table 3 , scaling an L1 on-chip above 128x8 is probably not a wise choice given the growing area requirements. However, as already mentioned in Section 4.4 , part of the ZBT SRAM could be used to hold the cache lines of a direct mapped L2 cache. In this scenario, only the tags would be held inside of the FPGA.

A sample cache organization for this approach could partition the 24-bit address into a 8-bit tag, 12-bit index and 4-bit block offset. The required tag RAM would be organized as 4096x8b, and would thus require 8 BlockSelectRAMs (in addition to those used for the on-chip L1 cache). The cache would have 4096 lines of 16 words each, and would thus require 64K words of the 128K words available in one of the ZBT SRAM chips. The cache hit/miss determination could occur in two cycles, with the data arriving after another two cycles. A miss going to PCI memory would take 66 cycles to refill a clean cache line and deliver the data to the initiator.

6 Related work

[19] gives experimental result for the cache-dependent performance behavior of 6 of the 8 benchmarks in the SPECint95 suite. Due to the *temporal configurability* we suggest for the MARC caches (adapting cache parameters to applications), they expect a performance improvement between 3% to 10% over static caches. [27] describes the use of the configurable logic in a hybrid processor to either add a victim cache or pre-fetch buffers to an existing dedicated direct-mapped L1 cache on an per-application basis. They quote improvements in L1 miss rate of up to 19%. [28] discusses the addition of 1MB of L1 cache memory managed by a dedicated cache controller to a configurable processor. Another approach proposed in [29] re-maps non-contiguous strided physical addresses into contiguous cache line entries. A similar functionality is provided in MARC by the pre-fetching of data into the FIFOs of streaming ports. [30] suggests a scheme which adds configurable logic to a cache instead of a cache to configurable logic. They hope to avoid the memory bottleneck by putting processing (the configurable logic) very close to the data. The farthest step with regard to data pre-fetching is suggested in [31], which describes a memory system that is cognizant of high-level memory access patterns. E.g., once a certain member in a structure is accessed, a set of associated members is fetched automatically. However, the automatic generation of the required logic from conventional software is not discussed. On the subject

of streaming accesses, [20] is an exhaustive source. The 'Block' register of our streaming ports was motivated by their discussion of overly long startup times for large amounts of pre-fetched data.

7 Summary

We presented an overview of hybrid processor architectures and some memory access needs often occurring in applications. For the most commonly used RC components (off-the-shelf FPGAs), we identified a lack of support for even the most basic of these requirements.

As a solution, we propose a general-purpose Memory Architecture for Reconfigurable Computers that allows device-independent access both for regular (streamed) and irregular (cached) patterns. We discussed one real-world implementation of MARC on an emulated hybrid processor combining a SPARC CPU with a Virtex FPGA. The sample implementation fully supports multi-threaded access to multiple memory banks as well as the creation of "virtual" memory ports attached to on-chip cache memory. The configurable caches in the current version can reduce the latency from 46 cycles (for access to DRAM via PCI) down to a single cycle on a cache hit.

References

1. Amerson, R., "Teramac – Configurable Custom Computing", *Proc. IEEE Symp. on FCCMs*, Napa 1995

2. Bertin, P., Roncin, D., Vuillemin, J., "Programmable Active Memories: A Performance Assessment", *Proc. Symp. Research on Integrated Systems*, Cambridge (Mass.) 1993

3. Box, B., "Field-Programmable Gate Array-based Reconfigurable Preprocessor", *Proc. IEEE Symp. on FCCMs*, Napa 1994

4. Buell, D., Arnold, J., Kleinfelder, W., "Splash 2 – FPGAs in Custom Computing Machines", *IEEE Press*, 1996

5. Rupp, C., Landguth, M., Garverick, et al., "The NAPA Adaptive Processing Architecture", *Proc. IEEE Symp. on FCCMs*, Napa 1998

6. Hauser, J., Wawrzynek, J., "Garp: A MIPS Processor with a Reconfigurable Coprocessor", *Proc. IEEE Symp. on FCCMs*, Napa 1997

7. Wittig, R., Chow, P., "OneChip: An FPGA Processor with Reconfigurable Logic", *Proc. IEEE Symp. on FCCMs*, Napa 1996

8. Jacob, J., Chow, P., "Memory Interfacing and Instruction Specification for Reconfigurable Processors", *Proc. ACM Intl. Symp. on FPGAs*, Monterey 1999

9. Triscend, "Triscend E5 CSoC Family", http://www.triscend.com/products/IndexE5.html, 2000

10. Altera, "Excalibur Embedded Processor Solutions", http://www.altera.com/html/products/excalibur.html, 2000

11. TSI-Telsys, "ACE2card User's Manual", *hardware documentation*, 1998

12. Koch, A., "A Comprehensive Platform for Hardware-Software Co-Design", *Proc. Intl. Workshop on Rapid-Systems Prototyping*, Paris 2000

13. Annapolis Microsystems, http://www.annapmicro.com, 2000

14. Virtual Computer Corp., http://www.vcc.com, 2000

15. Callahan, T., Hauser, J.R., Wawrzynek, J., "The Garp Architecture and C Compiler", *IEEE Computer*, April 2000

16. Li, Y., Callahan, T., Darnell, E., Harr, R., et al., "Hardware-Software Co-Design of Embedded Reconfigurable Architectures", *Proc. 37th Design Automation Conference*, 2000

17. Gokhale, M.B., Stone, J.M., "NAPA C: Compiling for a Hybrid RISC/FPGA Machine", *Proc. IEEE Symp. on FCCMs*, 1998

18. Koch, A., Golze, U., "Practical Experiences with the SPARXIL Co-Processor", *Proc. Asilomar Conference on Signals, Systems, and Computers*, 11/1997

19. Fung, J.M.L.F., Pan, J., "Configurable Cache", *CMU EE742 course project*, http://www.ece.cmu.edu/ ee742/proj-s98/fung, 1998

20. McKee, S.A., "Maximizing Bandwidth for Streamed Computations", *dissertation*, U. of Virginia, School of Engineering and Applied Science, 1995

21. Sun Microelectronics, "microSPARC-IIep User's Manual", http://www.sun.com/sparc, 1997

22. Weaver, D.L., Germond, T., "The SPARC Architecture Manual, Version 8", Prentice-Hall, 1992

23. PLX Technology, "PCI 9080 Data Book", http://www.plxtech.com, 1998

24. Xilinx, Inc., "Virtex 2.5V Field-Programmable Gate Arrays", http://www.xilinx.com, 1999

25. Xilinx, Inc. "Designing Flexible, Fast CAMs with Virtex Family FPGAs", *Xilinx Application Note 203*, 1999

26. Hennessy, J., Patterson, D., "Computer Architecture: A Quantitative Approach", *Morgan-Kaumann*, 1990

27. Zhong, P., Martonosi, M., "Using Reconfigurable Hardware to Customize Memory Hierarchies", *Proc. SPIE, vol. 2914*, 1996

28. Kimura, S., Yukishita, M., Itou, Y., et al., "A Hardware/Software Codesign Method for a General-Purpose Reconfigurable Co-Processor", *Proc. 5th CODES/CASHE*, 1997

29. Carter, J., Hsieh, W., Stoller, L., et al., "Impulse: Building a Smarter Memory Controller", *Proc. 5th Intl. Symp. on High. Perf. Comp. Arch. (HPCA)*, 1999

30. Nakkar. M., Harding, J., Schwartz, D., et al., "Dynamically programmable cache", *Proc. SPIE, vol. 3526*, 1998

31. Zhang, X., Dasdan, A., Schulz, M., et al., "Architectural Adaptation for Application-Specific Locality Optimizations", *Proc. Intl. Conf. on Comp. Design (ICCD)*, 1997