LOW-LATENCY HIGH-BANDWIDTH HW/SW COMMUNICATION IN A VIRTUAL MEMORY ENVIRONMENT

Holger Lange and Andreas Koch

Technische Universität Darmstadt
Embedded Systems and Applications Group (ESA)
Darmstadt, Germany
{lange,koch}@esa.cs.tu-darmstadt.de

ABSTRACT

Adaptive computers combine conventional software programmable processors with reconfigurable compute units. We present techniques that allow the high-performance realization of demand-paged, virtually addressed main memory shared between both of these processing elements. Furthermore, we have achieved low-latency communication between software running on the CPU and the reconfigurable compute unit, allowing even fine-grained hardware/software partitioning. A system-level evaluation quantifies the advantages of our approach.

1. INTRODUCTION

Reconfigurable compute units (RCU) have accelerated applications from a wide spectrum of domains [1]. In many cases, however, it has proven advantageous to combine conventional software programmable processors (CPU) with a reconfigurable compute unit. Such an architecture is often called an *adaptive computing system* (ACS). Its CPU executes performance noncritical or operating system functions, while the RCU implements the computation kernels of an application in a *hardware accelerator* (HA).

Programming an ACS still requires specialized knowledge in digital logic design and processor architecture. To ease development, considerable effort has been made in the creation of automatic compile flows [2, 4, 5, 6] targeting such heterogeneous computers. Ideally, such tools partition the application between the two kinds of processing elements (PEs). System-level performance is maximized when the communication overhead between the two partitions can be minimized. Generally, there are two classes of communications in such a setup. First, the data processed by the algorithms must be available to both PEs. This data is commonly located in main memory, inducing the need for high-performance high-bandwidth memory access for both CPU and HA. Secondly, control information is exchanged between the two PEs. Since only few data items are exchanged for this purpose, high-bandwidth transfers are not required. However, the communication latency for these exchanges now becomes a crucial factor and must be minimized.

The computation model, which orchestrates this interaction between heterogeneous PEs, is highly dependent on how well both of these communication requirements can be fulfilled. The work described here is motivated by the computation model used by the Comrade hardware/software compiler [6]. That model, which was presented in greater detail in [18], aims at fine-grained partitioning between CPU and HA execution. In this manner, operations unsuitable for hardware acceleration or requiring operating system services can be handled by quickly switching execution back to the CPU. Since the application data is kept in shared main memory, only a limited set of live variables has to be exchanged between HA registers and CPU variables. We will show that an ACS following this approach is efficiently implementable in today's technology. Since our fundamental techniques are not hardware-specific, they are widely applicable to other ACS platforms.

2. PLATFORM REQUIREMENTS

To guide the following discussion, we will now identify individual requirements on our hardware platform for actually implementing the communication mechanisms described above. The low-latency signaling and data communication between CPU and the HA will be referred to as **R1**. Efficient access to the shared memory requires a common address space (**R2**) to allow the exchange of pointers between CPU and HA, and high-throughput data transfers from and to memory for both PEs (**R3**). Software scheduling decisions made by the operating system *must* be respected in that the HA may not prevent the CPU from accessing memory (**R4**). This requirement is motivated by OS stability concerns: Critical functions, such as timekeeping and I/O, *must* be performed when demanded by the OS, otherwise data loss or a system crash may occur. Finally, the presence of an HA may not slow down the execution of software on the CPU (**R5**).

Practical solutions for most of these requirements have initially been described in [18]. In this paper, we describe significant advances over that work, specifically in the areas of **R1** to **R3**. This includes improvements of pre-existing features as well as the addition of new capabilities. As before, we take a system-level view: we consider operating system as well as computer architecture issues. All techniques are implemented and evaluated on a real hardware platform (a Xilinx ML310 board [13] based on a Virtex II Pro-FPGA [15] with two embedded PowerPC 405 processor cores), running a full-scale version of the multi-user, multi-tasking, virtual memory Linux operating system. Due to the increasing use of Linux even in the embedded world [7], such an architecture is of great practical interest.

3. PRIOR AND RELATED WORK

Prior work has often considered only a subset of these or other requirements. For example, [8] focuses on ease-of-use of CPU/HA communications by automatically mapping the HA registers to named files in the kernel /proc file system. While this removes the need for memory-mapping the hardware registers, the added file-system overhead increases latencies significantly and would violate **R1** of our model.

A similar approach is described in [9], but goes further by mapping the *entire* device structure at the configuration level (CLBs, BRAMs, etc.). While this allows the exchange of large chunks of data (by allowing the CPU direct access to the HA on-chip memories), the file-system integration again leads to latencies inacceptable for our execution model. None of these approaches considers the other requirements **R2** to **R5**.

[10] proposed a message-passing interface which would allow the HA master-mode access to memory (**R2**), and the underlying network-on-chip could be extended to provide **R4** (none of which is actually addressed by that work). The technique also does not deal with **R3** and **R5** at all.

For a completely different perspective on **R2**, the Philips SAA 7146A Multimedia Bridge [16], a chip used in many DVB systems, contains a simple MMU which can translate virtual addresses to physical page frame addresses if supplied with a single level page table by the associated device driver. However, this scheme is severely limited: First, the page tables are not automatically updated by the OS kernel whenever page allocations change, but by the device driver (introducing additional synchronization overhead). Second, the single level page table, stored in a dedicated 4 KB memory page, limits the virtual address space per DMA channel to a mere 4 MB. For our scenario of HA and CPU acting as equal peers, such a small shared memory could be realized more efficiently using the DMA buffer/AISLE approach of [18]. Finally, virtual address spaces in the Philips approach are *local* to each DMA channel (8 in total), complicating unified address handling even more.

R2 is also addressed by the dedicated *virtual memory window* described in [3]. In this implementation, HA and CPU access a virtually addressed shared memory area. The CPU is signalled by the HA when the latter attempts to access a page not present in the window, causing a virtual memory window manager in the OS to *copy* the missing page(s) between shared and main memory, thus allowing the HA full memory access. However, the technique is limited by the slow address translation (4 cycles per TLB access) and the high copying overhead (up to 50% of the execution time). Furthermore, the described implementation limits the window size to just 16 KB, making it unsuitable for the data-intense processing often performed by HAs.

4. HIGH-PERFORMANCE HA MEMORY ACCESS

As other reconfigurable SoCs, the Xilinx Virtex II Pro device uses on-chip busses to tie its different components together. The CoreConnect Processor Local Bus (PLB) [12] is normally used for memory accesses. However, both Xilinx implementations of the CoreConnect protocol (even the recent 128 bit version) as well as the DDR-DRAM controllers have significant limitations that render them unable to achieve the maximum performance. For our platform, that would be 64-bit DDR-200 operation, yielding a 1.6 GB/s transfer rate. These deficiencies are further discussed in [17, 18].

As a solution, we proposed the FastLane high performance memory interface (Figure 1) in [17]. It establishes a *direct* connection of the memory-intense HA to the central memory controller *without* an intervening PLB. By also using a specialized, lightweight protocol, we can avoid the arbitration as well as protocol overheads associated with PLB. This significantly reduces the latency between the HA core and the memory controller, since no wrapper logic is interposed between the two. Incidentally, this arrangement also allows the HA the use of the already existing PLB slave in the memory controller core for HA to CPU communication (providing the foundations for **R1**).

To improve upon the previous results, we have created *FastLane+*, enhanced over the original version by doubling its data path width to 128 bits, still clocked at 100 MHz, thus allowing full double data-rate operation. We achieved this by extending the existing DDR-DRAM controller of the EDK reference design [14].

Even with its improved performance, the new design still enforces OS scheduler decisions at the hardware level by never letting the HA starve the CPU from memory accesses (**R4**). To this end, the CPU and other bus master devices always have priority over the HA block, which can be explicitly designed to tolerate access delays. The arbitration

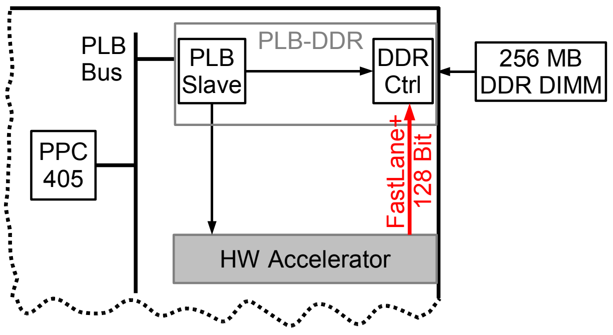

Fig. 1. FastLane+: Attaching HA directly to DDR controller

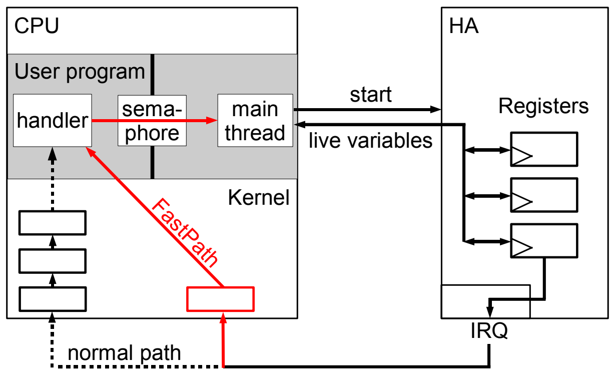

Fig. 2. FastPath: Low-latency SW calls and live variable transfers

logic required is completely hidden from the HA within the FastLane+ interface. The CPU, and other bus master devices, may interrupt memory accesses initiated by the HA at any time, while the HA cannot interrupt the CPU, and has to wait for the completion of a CPU-initiated transfer.

5. LOW-LATENCY CPU↔HA COMMUNICATION

Adaptive computing systems *combine* conventional CPUs and HAs to efficiently deliver high compute performance. In our model of computation [18], we allow the creation of HAs from code containing operations that cannot be efficiently mapped to reconfigurable logic. This might include, e.g., dynamic memory allocations or I/O such as a printf() function call. Normally, the presence of these operations in a computation kernel would prevent it from being realized in an HA. But if our hardware/software partitioning step determines that they occur sufficiently infrequently (based on dynamic profiling) the *rest* of the kernel is still realized as an HA. If these rare conditions actually arise at run-time, the HA requests execution of the hardware-infeasible operation as a *software service* on the CPU.

For high-performance, these switches should be performed with minimum latency (**R1**). This is easily achievable in an embedded system running either no operating system or only a lightweight RTOS. With the increasing complexity of embedded systems, there is a trend to run them under full-scale operating systems such as Linux [7]. Unfortunately, such OSs introduce a relatively long time penalty for switching from one task to another. Interrupt handlers, which are responsible for accepting requests from hardware devices such as the HA, also suffer from additional switching delays, making hardware/software execution switches costly and violating **R1**.

We have developed hardware/software mechanisms to achieve **R1** even in such a hostile environment. To put our work into perspective, consider that even when running a Linux version patched for low-latency, the interrupt response time on the ML310 platform is 62μs. The interrupt initiated by the HA passes through numerous layers in the Linux kernel before it reaches the handler in the software-portion of the ACS application. (shown as a dotted path in Figure 2). This long overhead would allow only a very coarsely granular hardware/software partitioning.

To reduce the latency, we let the HA communicate with the CPU using a *dedicated* interrupt vector of the PowerPC 405. This vector invokes a special handler that replies to all software service requests from the HA. If required, the execution flow (shown as a solid arrow in Fig. 2) can *directly* branch to a C-callback function in the user program, which has easy access to all application data (e.g., global variables etc.). By employing a dedicated handler, we can significantly reduce the interrupt overhead and thus work towards **R5**. The user-space interrupt handler is synchronized with the main software thread using a semaphore. This semaphore, however, is not realized using the comparatively heavyweight semaphore mechanisms in the C standard library. Instead, to avoid memory accesses and bus contention, it is implemented in a special CPU register not used by the software compiler. Thus, we avoid increasing pressure on the compiler's register allocator. The main thread is still controlled by the Linux scheduler, which may choose to suspend it, thus at first glance eliminating the advantage of the fast semaphore operation. While this could be avoided by increasing the priority of the main thread, such broad measures should only rarely be necessary, since the callback handler can execute latency-critical operations directly. As we will show in Section 7, these combined measures, which we call *FastPath*, significantly reduce response latency, allowing frequent hardware/software switches without increasing system load. Additionally, the interrupt response time is nearly independent of the system load. The new signalling scheme now allows the use of HAs to accelerate even shorter sections of the program.

Beyond the quick signaling between HA and CPU, **R1** also requires a low latency data exchange between HA hardware registers and CPU software variables. This is achieved by the CPU issuing reads and writes to the memory-mapped

HA registers (see Figure 2). At the hardware level, these are actually handled by the PLB slave the HA shares with the memory controller. The latency for a read is 20 ns, while a write takes 40 ns per data item. Since, in general, a software service requires the exchange of only very few variables [19], the overhead of these data transfers is negligible compared to the signaling latency.

6. HA INTERACTION WITH VIRTUAL MEMORY

The design of an adaptive computing *system* has to carefully consider both hardware and software aspects. One of these is the integration of the HAs with the operating system, the software environment shared by all programs running on the ACS. Since the HA must be capable of master-mode access to main memory, this is a non–trivial endeavor in an OS environment supporting virtual memory. On the CPU side, a memory management unit (MMU) is responsible for translating virtual *user space* addresses, as handled by the software applications, into physical bus addresses, which are sent out from the CPU via the PLB. In this manner, address translation and the resolution of page faults are transparent for application software. However, the HA does not have access to the processor's MMU.

6.1. AISLE

In [18], we have introduced the Accelerator-Integrating Shared Layout for Executables (AISLE) as one solution to this problem. In AISLE, all data areas of the program (stack, heap and initialized data segments) are kept *inside* a so-called direct memory access (DMA) buffer. Such a DMA buffer is guaranteed to consist of contiguous physical memory pages that are always present in physical RAM, they will never be swapped out to disk. Virtual addresses, located in the DMA buffer, always have a constant offset from their physical address. By transparently compensating for that offset when initiating memory accesses from the HA, the HA can now also use virtual addresses. Thus, pointers can be freely exchanged between both CPU and HA. As an aside, since the PowerPC 405 has no multi-processor cache-coherency mechanisms (like most embedded processors), keeping the cache coherent between CPU and HA execution also requires software intervention. We realized this in a fashion that allows the CPU to execute at full speed (**R5**).

While AISLE fulfills all requirements, it does have limitations. The size of the DMA buffer is normally set at the start of a program. At runtime, it can be resized only with significant overhead. Thus, it is always allocated to the largest data area required by the program, which can be wasteful. Additionally, DMA buffers are always present in physical memory and remove their areas from the general demand-paging performed by the OS. This not only reduces

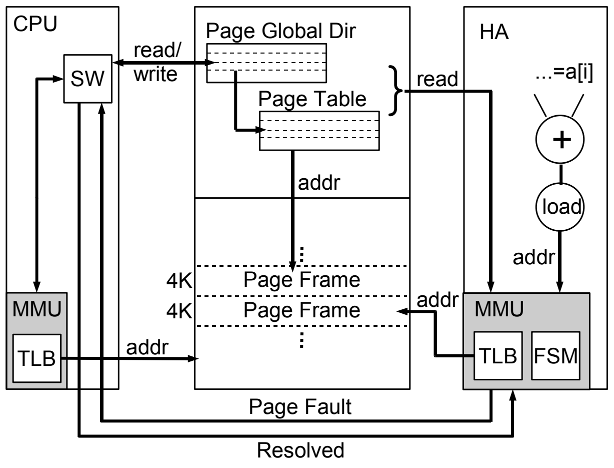

Fig. 3. PHASE/V TLB system: HA↔OS interactions

the amount of memory available to the entire system, but also forces the loading of all data areas in the program's executable file, even if they are not actually required at runtime.

6.2. Full Virtual Memory Support in the HA

As an alternative solution, we will now describe the new Processor-Hardware Accelerator Shared Environment with Virtual Addressing (PHASE/V). Here, the HA is integrated with the MMU-based virtual memory system. Instead of mapping just a *window* of the virtual address space in the form of a DMA buffer, we map the *complete* virtual address space between HA and CPU. All virtual memory features, such as demand paging, swap space, copy-on-write, and file-backed `mmap` mappings, are thus supported both in hardware and software. Since demand paging is now available for the HA, memory pages are physically allocated (and loaded from the executable file) only when they are actually needed, not in advance.

Like many embedded processors, the PowerPC 405 does not allow external access to its MMU (Figure 3). Thus, we had to implement a *separate* translation lookaside buffer (TLB) in the HA. It operates in a direct-mapped fashion and has 64 entries. Beyond the HA-TLB, an associated FSM is able to walk the *CPU MMU-managed* page tables stored in main memory, consisting of the Page Global Directory and the Page Tables themselves. This FSM is responsible for performing a virtual-physical address translation in case a HA-initiated memory access leads to a HA-TLB miss. Should the mapping be already present in the HA-TLB, the translation only takes a single clock cycle, providing the HA with maximum throughput at minimum latency (**R3**).

In addition to performing virtual-physical address translations, our scheme must also be able to handle page faults. These occur when the HA requests a virtual address that does not yet have (or no longer has) an associated page in physical memory, a condition that will be detected during the HA-initiated page-table walk. For its resolution, we rely on the standard OS mechanisms: We use our FastPath signaling scheme introduced in Section 5 to request the handling of the page fault as a software service. The Linux kernel then fetches the missing page frame, updates the page tables, and switches back to the HA to continue processing. To treat the case when the OS flushes page frames from memory, or swaps them out to disk, we have modified the OS kernel to not only invalidate the CPU-TLB, but also the HA-TLB, which is visible to the CPU in a memory-mapped fashion. Relying only on sniffing for CPU TLB- or page table writes is insufficient, since not all HA-accessed pages will have been mapped into the kernel page tables before starting the HA. As AISLE, PHASE/V also uses software to ensure cache-coherency between CPU and HA. PHASE/V does support multiple HAs, either sharing the same HA-TLB or having dedicated TLBs (which would require explicit inter-HA-TLB coherency mechanisms).

With PHASE/V, the HA now has the same capabilities as the CPU for virtual memory management. However, as we will examine in the next Section, these features do have a performance penalty over the simpler AISLE approach.

7. EVALUATION

In this section, we evaluate all of these techniques together at the system level.

FastLane+ has significantly improved performance by exploiting the full memory bus width, almost doubling the throughput over FastLane and achieving $\approx 90\%$ of the theoretical maximum for this memory. As before with FastLane, running an HA in parallel to normal software programs slows down the CPU only *negligibly* (**R4**). Incidentally, despite being focused on the HA, the improvements in FastLane+ even result in minor software speedups (up to 5%) for the CPU. A more detailed discussion of this methodology is found in [17].

To evaluate our low latency hardware/software communications scheme, we consider the impact of its delay $t_{overhead}$ on the effective speedup achievable, given the raw hardware acceleration factor HW_{accel} and the execution time t_{HA} spent in hardware before switching back to software, e.g., to request a service.

$$\text{effective speedup} = \frac{t_{SW}}{t_{HA} + t_{overhead}} = \frac{t_{HA} \times \text{HW}_{accel}}{t_{HA} + t_{overhead}}$$

The delay time has two components: the interval t_{IRQ} between the HA initiating an interrupt and the reaction in the user space interrupt handler (e.g., determining the interrupt cause by reading from an HA register), and the time t_{sem} between the handler accessing the semaphore and the resumption of the main program thread. Measured times

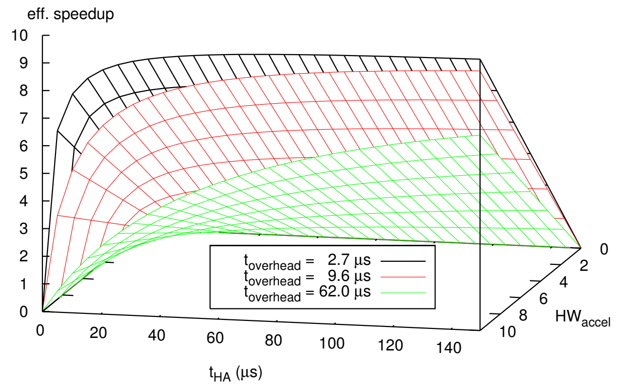

Fig. 4. Effective speedup as function of HA execution time and raw HW acceleration factor for different latencies

are cycle accurate, being determined by a dedicated hardware counter. As baseline, we measured the overhead for the standard Linux interrupt path on the ML310 (employing the usual tasklet-driven wait queue in lieu of our fast semaphore), resulting in $t_{overhead} = 62\mu s$. In contrast to this, FastPath achieves a $t_{overhead}$ between just $2.7\mu s$ and $9.6\mu s$ (best case: $t_{IRQ} = 2.1\mu s$, $t_{sem} = 0.6\mu s$; worst case: $t_{IRQ} = 8.8\mu s$, $t_{sem} = 0.8\mu s$). Combined with the negligible times for the live variable transfer (20…40ns, see Section 5), we can thus shorten the total overhead by a factor of 6.5x to 23x (**R1**). In this manner, FastPath even outperforms specialized real-time variants of Linux, such as RTAI/LXRT or tuned versions of recent 2.6.x kernels, running on a desktop PC CPU much faster than our 300 MHz PowerPC 405 [11].

The system-level effects of these measurements are visualized in Figure 4. It shows the achievable effective speedup (z-axis) for different combinations of raw hardware speedup (y-axis), execution time spent in hardware (x-axis), and the different communication overheads. For the latter, the bottom surface shows the effective speedup achievable using the standard Linux communications mechanism, while the upper two surfaces show the impact of FastPath (top: best case, middle: worst case). It is obvious, that for very short hardware execution times, the communication overhead dominates and even high hardware acceleration factors yield only small effective speedups or even slowdowns. All three surfaces in the figure converge against an imaginary plane representing the theoretically optimal speedup (where eff.speedup= HW_{accel}). The two FastPath surfaces already reach 90% of the optimum after just $23\ldots75\mu s$ of hardware execution, while the conventional communication mechanisms requires much longer times spent in hardware to achieve similar effective speedups. Such information is crucial to perform a high-quality hardware-software partitioning, since it will directly influence the granularity of the execution sections assigned to hardware or software processing.

Both AISLE and PHASE/V allow the free interchange

List length	SW	HA with AISLE	HA with PHASE/V
16K	7.4 ms	3.4 ms	3.5 ms
32K	15.8 ms	6.8 ms	12.2 ms
128K	68.3 ms	27.4 ms	64.0 ms

Table 1. Runtimes of the pointer-handling application

of userspace address pointers between the HA and CPU, which allows much faster operation than the explicit copying of data between CPU and HA memories [18]. However, the full virtual memory capabilities of PHASE/V do come at a performance cost. Table 1 shows the execution times for processing a different number of elements of a linked list randomly arranged in memory. Since we are interested only in quantifying the memory access overhead, we perform only a trivial operation on the data to avoid measuring the effect of possible hardware acceleration [18]. The results show that the performance of PHASE/V is highly dependent on the size of the application's data set. In the 16K case, the page mappings for the memory area completely fit into the HA-TLB, thus no threshing occurs and the performance roughly equals that of AISLE. With 32K list elements, the performance drops since TLB threshing begins to occur. Still, PHASE/V is 23% faster than the SW version. At 128K, heavy TLB threshing slows the acceleration. However, even in this extreme case, PHASE/V is still able to outperform the pure SW implementation by 6%. Thus, depending on the application requirements, the simpler AISLE should continue to be used if the enhanced capabilities of PHASE/V are not needed.

8. CONCLUSION AND FUTURE WORK

In our aim to demonstrate the practical feasibility of adaptive computing systems supporting fine-grained application partitioning between CPU and HA, we have achieved both improvements to previous work as well as the realization of completely new capabilities.

We have further improved the interface between main memory and HA, with FastLane+ almost reaching the theoretical throughput of the memory chips themselves (and significantly exceeding that of the CPU). Beyond the memory access itself, we have also further considered how CPU and HA can interact in a virtual memory environment. With our new PHASE/V technology, we have shown that full demand paging *is* possible even if the CPU-MMU is not directly accessible to the HA. Apart from memory throughput, the effective speedup achievable for a finely-partitioned application is also highly dependent on the delay of inter-partition communications. Using the new FastPath low-latency communications mechanism, even shorter fragments of application code can now be successfully hardware-accelerated.

One area of future work is increasing the efficiency of PHASE/V. Even using FastPath, the explicit synchronization of the TLBs in the CPU and the HA does have a relatively high overhead. This could be avoided by having a shared TLB between CPU and HA. The CPU-internal TLB could be switched off and the shared TLB would be implemented in reconfigurable logic outside of the CPU hardcore. Since we have shown that it is possible to construct such TLBs running at full bus speeds on the FPGA, this design would not impose additional performance penalties.

9. REFERENCES

[1] Gokhale M., Graham P.S., "Reconfigurable Computing", Springer, 2005

[2] Budiu M., Venkatarami G., et al., "Spatial Computation", Proc. Intl. ACM Conf. on ASPLOS, 2004

[3] Vuletić M., Pozzi L., Ienne P., "Virtual Memory Window for Application-Specific Reconfigurable Coprocessors", Proc. Design Automation Conference (DAC), San Diego, 2004

[4] Callahan T., Hauser J., Wawrzynek J., "The Garp Architecture and C Compiler", *IEEE Computer*, 04/2000

[5] MacMillen D., "Nimble Compiler Environment for Agile Hardware", Storming Media LLC (USA), 2001

[6] Kasprzyk N., Koch A., "High-Level-Language Compilation for Reconfigurable Computers", Proc. Intl. Conf. on Reconfigurable Communication-centric SoCs (ReCoSoC), 2005.

[7] Balacco S., "Linux in the Embedded Systems Market (Vol. VII)", Venture Development Corp, 2007

[8] So H., Tkachenko A., and Brodersen R., "A Unified Hardware/Software Runtime Environment for FPGA-Based Reconfigurable Computers using BORPH", Proc. 16th Int. Conf. on Field Programmable Logic and Applications (FPL), Madrid, 2006

[9] Donlin A., Lysaght P., Blodget B., and Troeger G., "A Virtual File System for Dynamically Reconfigurable FPGAs", Proc. 14th Int. Conf. on Field Programmable Logic and Applications (FPL), Antwerp, 2004

[10] Nollet V., Coene P., Verkest D., Vernalde S., and Lauwereins R., "Designing an Operating System for a Heterogeneous Reconfigurable SoC", Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), Nice, 2003

[11] Laurich P., "A comparison of hard real-time Linux alternatives", LinuxDevices, 2004

[12] IBM, "The CoreConnect Bus Architecture", *White Paper*, 1999

[13] Xilinx, "ML310 User Guide" *(UG068)*, 2005

[14] Xilinx, "Embedded System Tools Reference Manual" *(UG111)*, 2006

[15] Xilinx, "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet" *(DS083)*, 2005

[16] Philips Semiconductors, "SAA7146A Multimedia bridge, high performance Scaler and PCI circuit (SPCI)", Product Specification, 2004

[17] Lange H., Koch A., "Design and System Level Evaluation of a High Performance Memory System for reconfigurable SoC Platforms", Proc. HiPEAC Workshop on Reconfigurable Computing, Ghent, 2007

[18] Lange H., Koch A., "An Execution Model for Hardware/Software Compilation and its System-Level Realization", Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL), Amsterdam, 2007

[19] Kasprzyk, N., "COMRADE – Ein Hochsprachen-Compiler für Adaptive Computersysteme", Ph.D. thesis, Tech. Univ. Braunschweig, 2005