Architectures and Execution Models for Hardware/Software Compilation and their System-Level Realization

Holger Lange, Andreas Koch, *Member, IEEE*

Abstract—We propose an execution model that orchestrates the fine-grained interaction of a conventional general-purpose processor (GPP) and a high-speed reconfigurable hardware accelerator (HA), the latter having full master-mode access to memory. We then describe how the resulting requirements can actually be realized efficiently in a custom computer by hardware architecture and system software measures. One of these is a low-latency HA-to-GPP signaling scheme with latency up to 23x times shorter than conventional approaches. Another one is a high-bandwidth shared memory interface that does not interfere with time-critical operating system functions executing on the GPP, and still makes 89% of the physical memory bandwidth available to the HA. Finally, we show two schemes with different flexibility / performance trade-offs for running the HA in protected virtual memory scenarios. All of the techniques and their interactions are evaluated at the system level using the full-scale virtual memory variant of the Linux operating system on actual hardware.

Index Terms—reconfigurable computing, FPGA, hardware accelerator, memory system, operating system integration, virtual memory

1 Introduction

Reconfigurable devices acting as *hardware accelerators* (HA) can improve pure compute performance as well as efficiency (e.g., power requirements and integration density) [1]. Despite these advantages, practical use of the technology is still uncommon, generally due to a programming environment which is unfamiliar to conventional software developers.

For limited application domains such as digital signal processing, more accessible flows that employ established tools and notations (e.g., MATLAB, Simulink) do exist [2], [3], but the automatic translation of traditional software (SW) high-level languages such as C or Java to exploit a reconfigurable device as accelerator is still uncommon and hampered by tool limitations. Such restrictions often include the prohibition of pointers or conditionals in loops [4], [5], or require user-specified annotations beyond the core language [6]. If the input program does not completely match the tool limitations, translation to an HA will be aborted.

To ease the transition from a SW-programmable general-purpose processor (GPP) to hardware (HW) acceleration, some flows [7], [8], [9], [10] target *adaptive computing systems* (ACSs): combinations of a GPP and HA, with both *processing elements* (PEs) acting as equal peers. On an ACS, parts of an input program with low instruction-level parallelism, highly irregular control flow, or area-intensive floating point computations can remain in SW on the GPP, to which the compiler can also fall-back if it encounters a language construct it cannot (yet) map to an accelerator. The program always remains executable and the developer can be advised of the problematical sections to incrementally rewrite them as desired for full acceleration.

The partitioning of the input program between the different PEs, both at compile time as well as in its runtime implementation, is a crucial issue for an ACS. Due to the large body of prior research in the general field of HW/SW partitioning, we will focus on the latter issue in this work: How the overhead of co-executing the application on the different PEs can be minimized by appropriate HW architectural and operating system-level measures. As we will discuss later, key questions in this context are the efficient access of multiple PEs to shared in-memory data structures and inter-PE signaling. Their answers will directly influence the choice of the *execution model*, the set of rules and protocols that orchestrates the co-execution of heterogeneous PEs.

Such an execution model is essential when implementing an automatic tool flow that targets an ACS: The HW/SW compiler requires a precise model which defines the interaction between the different PEs as well as shared components such as memory. This differs significantly from the freedom that is enjoyed when manually designing HW architectures, which allows the mixing-and-matching of different paradigms. Since the compiler always adheres to the rules that are imposed by the model, the execution model determines the available solution space for the automatically compiled architectures.

Reconfigurable Systems-on-Chip (rSoC) that combine different components (including heterogeneous PEs) connected by standard buses [19], [20], [21] on a single reconfigurable device can also be used as an ACS. We will discuss the widespread IBM CoreConnect bus [21] as an example.

The main contributions of this work are:

- An *execution model* which supports a fine-grained division of labor between a GPP and an HA that also allows the latter to call SW functions for HA-unsuitable or infrequent

- *H. Lange is with the LOEWE Research Center AdRIA, A. Koch is with the Embedded Systems and Applications Group, both at Technische Universität Darmstadt, Germany.*

Manuscript received April 99, 9999; revised January 99, 9999.

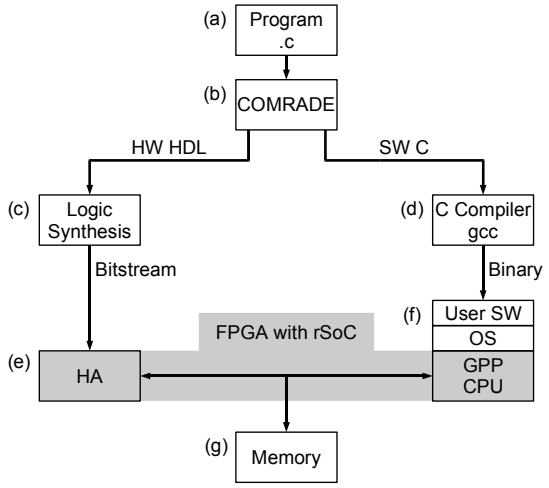

Fig. 1. System overview diagram

operations. Actual use of this model requires certain special capabilities from a target platform. We present practical solutions to all of them next.

- A low-latency *communication scheme* between GPP and HA that supports both efficient signaling and live variable exchange.
- High-performance *memory access* for the HA which exploits the full transfer rates of the physical memory.
- Robust, secure and efficient *system integration* of an HA in a multi-tasking protected virtual memory environment. To this end, we apply both HW architecture as well as operating system measures.

All of our techniques have been evaluated in an actual HW prototype which runs a full-scale version of Linux (including MMU support).

Figure 1 gives an overview of our system architecture and design flow. An input C program (a) is processed by our compiler COMRADE (b) [10], which partitions it into SW (exported as C and compiled using a conventional SW compiler, d) and HW parts (as Verilog HDL). The latter are subsequently processed by logic synthesis (c) and mapped into an FPGA. This FPGA, being an rSoC, contains the GPP (f, which executes the operating system and the SW parts of the compiled application) and the HA that corresponds to the HW parts (e). The FPGA also implements the communication between the HA, the GPP, and external memory (g).

2 Execution Model

The four execution models discussed in this Section aim to support the compilation from C to a GPP/HA combinations. One of them [7] targets ASICs, the other three [8], [9], [10] target reconfigurable ACSs. However, the difference in target technology does not affect the following discussion. Of greater interest are the different granularities of the HW/SW partitioning supported by the models.

ASH [7] can switch between GPP and HA only at procedure boundaries. However, once an entire procedure has been been moved to the HA, it may call SW functions that are executed on the GPP (see Figure 2). Procedures form natural partitioning

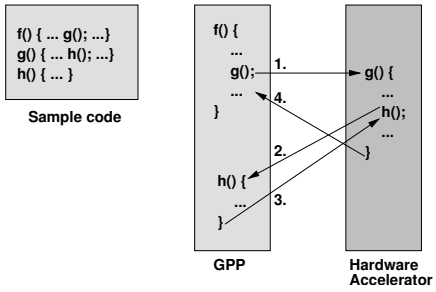

Fig. 2. ASH execution model

```
...
u = (int) sqrt(a + b);
v = c - d;
for (n=0; n<1000; ++n, p=p->next) {
  v += u;
  if (v > 10000) {
    printf("warning: v too large, rescaling");
    v *= 0.271844;
  }
  p->val = v;
}
w = 53 * v;
...
```

Fig. 3. Sample program with HA-unsuitable statements

delimiters, but the presence of only a single construct in the C source code that is not amenable to HW compilation (e.g., an operating system call or floating-point operation) will prevent the acceleration of the *entire* procedure, even if the problematical operation would occur only rarely (if ever) during an actual program execution.

The execution models of GarpCC [8], Nimble [9] and COMRADE [10] support a finer partitioning granularity which allows switches even *within* procedures. Since the partitioning is performed based on dynamic profiling, slow-downs due to excessive GPP/HA switches and their associated communication overhead can be avoided (see [10] for more details).

The code fragment shown in Figure 3 will be used to guide the following discussion. The `for` loop is surrounded by statements which are better left on the GPP (e.g., having low

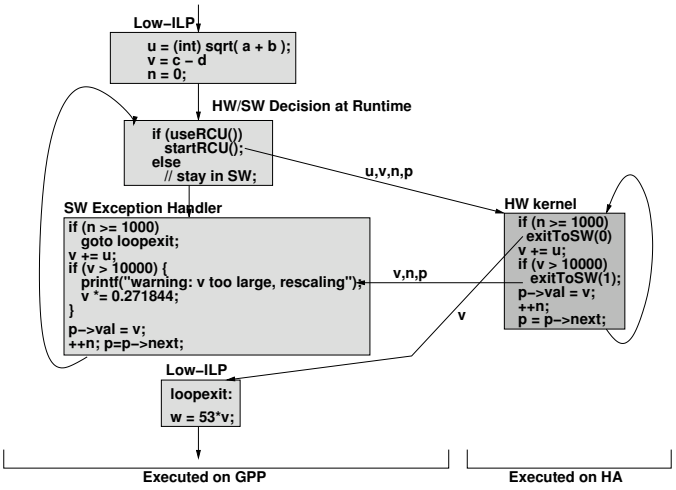

Fig. 4. Example in the Nimble execution model

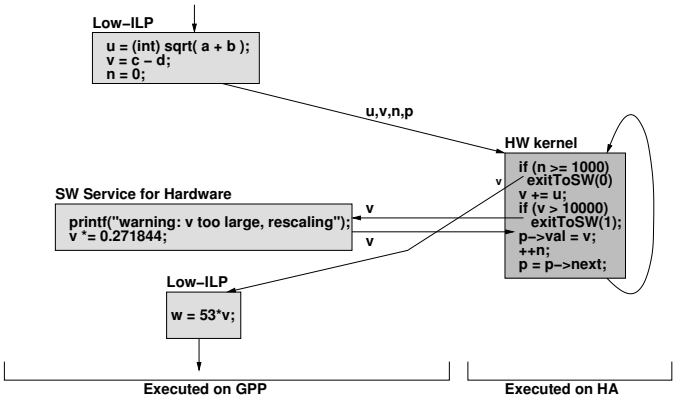

Fig. 5. Example in the COMRADE execution model

ILP or being only infrequently executed). Library functions and floating point-operations are assumed to be unsuitable for direct HW compilation.

With the fine-grained partitioning supported by GarpCC, Nimble, and COMRADE, it is possible to move just the loop to the HA as a so-called *kernel*, while the initial and trailing statements are left on the GPP. Based on dynamic profiling, the compilers might recognize that the exceptional (and HA-unsuitable) branch $v > 10000$ is executed only very rarely in practice, and thus HA-execution of the rest of the loop remains profitable. Once the condition occurs, however, it must be handled as specified by the input program: Execution switches back to the GPP, the live variables (shown as edge labels in Figures 4 and 5) are retrieved from HA registers to their corresponding SW variables, and the body of the `if` is executed in SW. Note that the transfer of the live variables occurs under GPP control, the HA acts as a *slave* in this process.

The three compilers differ in how the generated HW/SW continues to execute *after* switching to the GPP and executing the HA-unsuitable code. GarpCC and Nimble proceed to execute the rest of the current loop iteration in SW before they consider switching back to the HA. Our new COMRADE execution model (Figure 5) allows switching back *immediately* after completing the HA-unsuitable code. In the example, only the `printf` and the floating-point multiply execute on the GPP as a so-called *SW Service*. Such fine-grained switching requires low-latency GPP-HA communication from the ACS platform (discussed in Section 6), but also reduces the number of live variables that need to be exchanged between GPP and HA (due to the limited scope of a SW Service compared to an entire loop body). A kernel running on the HA can access multiple Services in this manner.

General-purpose C programs often make heavy use of pointer operations. When they are to be translated to an ACS, GPP-HA interaction goes beyond plain signaling and slave-mode live variable exchange. Now, pointer-based data structures must be operated on by both PEs. This requires free exchange of addresses and compatible address arithmetic as well as the ability of the HA to autonomously access main memory (in so-called *master* mode), preferably at high bandwidths. All of these issues, also encompassing virtual memory, will be discussed in the next Sections.

TABLE 1
Summary of Platform Requirements

Requirement	Description
LOWLAT	Low-latency GPP↔HA communication
ADDRESS	Shared GPP↔HA address space
HAMEM	High-throughput HA memory access
PROTSYS	HA access cannot affect other processes
PROTCODE	software code protected from HA access
OSSCHED	HA must obey OS scheduler
SWPERF	HA may not slow down software

3 Platform Requirements

The core of this work lies in explaining how to design and implement an ACS HW/SW architecture that supports the fine-grained COMRADE execution model. To structure this discussion, we will first identify individual requirements and assign them unique names (set in **bold type**), which will be referenced later in the detailed implementation descriptions.

GPP-HA signaling, which is needed in all execution models discussed above, should have low-latency (**LOWLAT**), but does not require high-throughput (only few live variables will generally need to be exchanged in a good partitioning).

The use of pointers by both PEs requires a common address space (**ADDRESS**), possibly achieved by sharing main memory or even parts of the cache hierarchy. Furthermore, it demands an HA that is capable of GPP-independent high-throughput access to main memory (**HAMEM**). The latter is lacking from many real ACS platforms. At best, the HA can often only access dedicated memories that need to be explicitly initialized by the GPP, which also has to retrieve the result data after the HA has completed its computation. This need for explicit allocation of HA-accessible memory as well as the copying of data significantly complicates the handling of pointer-based algorithms (see [38] for a more detailed discussion).

With growing application complexities, virtual protected memory is becoming more important even in embedded scenarios. Supporting **ADDRESS** and **HAMEM** becomes more difficult in such an environment. For security, **HAMEM** should have the HA restricted to access only the memory of the process that runs the SW part of the hybrid HW/SW application, the rest of the system's memory spaces must not be accessible (**PROTSYS**). For even finer grained memory protection, the executable code of the HW/SW application process itself should also be inaccessible to a (potentially rogue) HA (**PROTCODE**).

Finally, the performance of SW on the ACS should not be impeded by the HA. This encompasses that the HA must respect operating system scheduling decisions (**OSSCHED**), e.g., the HA has to be prevented from starving the GPP from memory accesses that are urgently needed to handle network packets in time. We also want to avoid slowing down the SW part of the HW-accelerated application (**SWPERF**).

We now examine in greater detail how these requirements have been addressed in prior work and how they can be achieved on current platforms. Our focus will be on using FPGAs as reconfigurable target devices for the HA, and Linux, which continues to grow market share in embedded scenarios (

[11], market share according to [35]: 20% Linux vs. VxWorks 9% and Windows 12%), as an example for a full-scale multi-tasking virtual protected memory operating system.

4 Prior and Related Work

Prior research on ACS architecture often considers only a subset of the above requirements. For exchanging live variables, [12] emphasizes ease-of-use by automatically mapping the HA registers to individually named files in the Linux /proc file system. However, routing each access through the file system increases latencies significantly and violates our **LOWLAT** requirement for GPP-HA communication.

The approach of [13] goes even further: The *entire* device structure (described at the configuration level of CLBs, BRAMs, etc.) is mapped into the file system. While multi-word chunks of data can now be exchanged between GPP and HA more efficiently by copying them to and from on-chip memories of the FPGA, the file system overhead still makes frequent live variable transfers too costly (violating **LOWLAT** again). Neither of these two techniques addresses our other requirements.

An approach that attempts to achieve **ADDRESS** is the message-passing interface of [14]. While not described in that work, it appears to be possible to extend the underlying network-on-chip to implement **PROTSYS**, **PROTCODE**, and **OSSCHED**. However, the **HAMEM** and **SWPERF** capabilities that we require are not dealt with at all.

A very promising ACS architecture [15] replaces one or more of the GPPs in a Symmetric Multi Processing platform with a reconfigurable device, which is now connected directly to the processor bus (e.g., the Intel Front Side Bus). While such an architecture can theoretically achieve low latency and high bandwidth at the HW level, the programming interface appears to be only rudimentary: A central controller orchestrates the I/O for one or more HAs that operate only as slaves on the same reconfigurable device. The GPPs and the HAs exchange data via a statically mapped and locked-down shared region of main memory. The HA can signal SW that is running on a GPP via an interrupt, handled there by a callback function. Since no measurements on performance or latency are publically available, it is unknown to which degree the requirements of our execution model would be satisfied on such a platform in practice.

It is also worthwhile to look beyond adaptive computer architectures to see how some of our stated requirements have been addressed in the past. For example, the Philips SAA 7146A Multimedia Bridge [25] is a chip which is used in Digital Video Broadcast (DVB) standard receivers. Each of its eight DMA channels can be supplied with a single-level page table, managed by its associated device driver. It can thus translate virtual to physical addresses independently from the GPP. Such a mechanism would satisfy **ADDRESS** even in a virtual memory environment. However, the mechanism has a number of practical limitations: First, since the device driver sets up a separate set of page tables, page allocations that are performed by the operating system must be explicitly synchronized to those of the chip (incurring additional overhead). Second, only a single 4 KB page held in dedicated memory is available for the chip's page table, limiting the virtual address space to just 4 MB. Finally, the per-channel nature of each page table requires even more synchronization overhead when aiming for a unified address space across all channels.

[26] also attempts to solve the **ADDRESS** requirement for an ACS, here using a dedicated *virtual memory window* which is shared between GPP and HA. The HA signals the GPP on a page fault, causing the SW virtual memory window manager in the OS to *copy* the missing page(s) from main memory to the window (and vice versa). While the technique does satisfy **ADDRESS**, it has high overhead (slow address translation and numerous copy operations) and is limited to just a 16 KB window (much too small for the data-intense processing for which an HA could be especially beneficial).

[27] presents a simulation of virtual memory integration at cacheline granularity for multiple HAs. Although **ADDRESS** is satisfied here as well, the relatively small (16 entry) translation lookaside buffer (TLB) for the HA relies entirely on SW management, which imposes an unacceptable overhead except for small per-application virtual address working sets. Moreover, a real world system evaluation is missing.

As a side effect of implementing increasingly complex GPPs on FPGAs, the GPP memory management units (MMUs) must also be ported [28]. The OpenSPARC T1 uses a MMU comprising a TLB with 64 entries, which had to be reduced to 16 or 8 to fit on an FPGA [29]. HA-MMUs require less area, since they can rely on the protection mechanisms of the GPP-MMU/OS *combination* to satisfy **PROTSYS** and **PROTCODE**. Hence, GPP- and HA-MMUs are not directly comparable.

The prior and related work addresses some of the requirements we identified (Table 1 in Section 3). It can be roughly classified into two non-disjunct sets: One set [12], [13], [25], [26], [27] already has fundamental limitations in individual requirements. The second one [12], [13], [14], [15], [29] implements only a subset of our functional requirements. Hence, none of these approaches fully supports any kind of tightly-coupled execution model. In contrast, we propose high-quality solutions to individual requirements and combine all of these techniques in a well-balanced manner to improve overall system performance.

5 Target Platform

In order to avoid the complexity (and potential inaccuracies) of simulating a multi-PE system including memories and I/O devices, we evaluate our techniques on an actual HW platform.

We use the Xilinx ML310 embedded system development platform (see Fig. 6) as base for our ACS. Architecturally, it resembles a standard PC, with a variety of peripherals (USB, NIC, IDE, UART, AC97 audio, etc.) that are attached via a Southbridge ASIC to a PCI bus. However, the traditional CPU and Northbridge ASIC have been replaced by a Xilinx Virtex II Pro (V2P) FPGA [23]. This reconfigurable device embeds two IBM PowerPC 405 GPPs (internally running at 300 MHz) in an array of reconfigurable logic. Each of the GPPs has 16 KB I- and D-caches (2-way, 8 words/line, write back) and an MMU that comprises a 64-entry unified TLB (plus 8 data and 4 instruction shadow entries, page table walks done in SW).

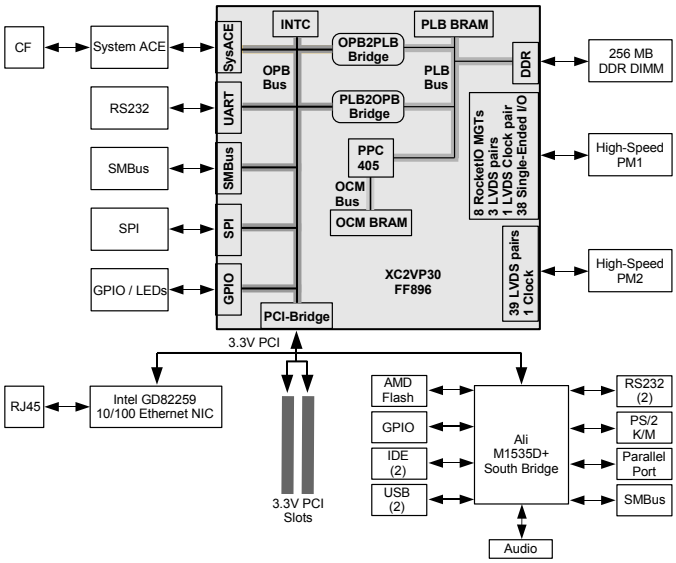

Fig. 6. ML310 system (from Xilinx manuals)

The entire compute architecture (GPPs, HAs, buses, memory interface) can be flexibly reconfigured for our experiments and achieves 100 MHz system clock rates in the reconfigurable array even for complex designs.

The vendor-provided initial reference design for the ML310 is shown on the gray background in Fig. 6. It uses a single PowerPC core (the second one remains inactive) to which several on-chip peripherals are attached using CoreConnect [21] buses (see [37] for more information). We will describe some concepts in greater detail below as required.

To stress-test our proposed solutions, we run the platform under the full-scale multi-tasking, virtual memory variant of Linux, instead of using a light-weight RTOS (which would impose a smaller and more deterministic load pattern on the internal buses).

While we used a concrete platform to verify our techniques, it is important to note that neither our architectural concepts nor their actual implementations are specific to the ML310. Instead, they should be portable to all platforms with similar characteristics, e.g., rSoCs based on GPP soft cores as well as more recent boards using newer Virtex-4 FX and -5 FX FPGAs as reconfigurable devices.

5.1 Vendor Flow for rSoC Composition

Assembling systems-on-chip (even disregarding reconfigurability) is a complex endeavor, generally due to the possibly large number of components that may also have disparate interfaces. While some interface standards do exist (such as the CoreConnect PLB discussed below), components often use different conventions internally and are generally connected to the system via protocol-converting *wrappers*, which sometimes induce a loss of performance.

The vendor rSoC composition flow Xilinx EDK [22] relies on PLB for interconnecting components such as the GPP and HA (shown in Figure 7). PLB has the following key characteristics, with those added by the recent version 4.6 (which is used on current Virtex-4 FX and -5 FX FPGAs) set in *italics*:

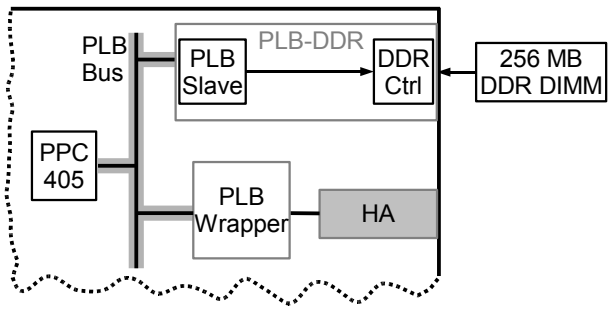

Fig. 7. HA integration via PLB

TABLE 2
PLB specification vs. actual implementation

	IBM PLB Spec v3.4	IBM PLB Spec v4.6	Xilinx V2P PLB
Clock	133 MHz	183 MHz	100 MHz
Address pipelining	2 cycles	unlimited	2 cycles
Latency	2 cycles	3 cycles	4 cycles
Burst length	unlimited	unlimited	16 words
Burst termination	anytime	anytime	full length

- Single beat transfers (one data item per transaction)
- Burst transfers up to 16 data words
- Cache line transfers (one cache line in 4 *resp. 8* data beats, cache-missed word first)
- Master/slave (self/peer initiated) transfers
- Atomic transactions (bus locking)
- Split transactions (separate masters/slaves performing simultaneous reads/writes)
- Central arbiter, but master is responsible for timely bus release (*no arbitration in point-to-point mode*)
- 64 / *128* bit wide operation

It is obvious that PLB is a feature-rich bus which aims for high-performance operation, but it carries a significant protocol complexity. This often requires conversions between the specialized internal protocols of a component and its general PLB interface, leading to “thick” wrappers.

Our concrete scenario is the efficient interconnection of the GPP, the HA, and the memory according to the requirements of the COMRADE execution model. In the vendor-suggested rSoC architecture, the HA accesses the memory (specifically, the DDR-DRAM memory controller) via PLB through two latency-inducing wrappers: A master-mode wrapper at the HA (which can initiate bus transactions) and a slave-mode wrapper at the memory controller (which just processes incoming requests).

5.2 Practical Limitations

The high performance which is theoretically possible with PLB is difficult to achieve in practice. For example, CoreConnect [21] specifies unlimited PLB burst lengths, with arbitrary burst termination and deep address pipelining. Actual PLB implementations, such as those used in three generations of Xilinx rSoCs, are more limited. For the Virtex II Pro devices which are employed in our ML310 ACS, some specifics are given in Table 2.

TABLE 3
Area overhead for bus wrappers in vendor flow

Wrapper for	Min. Size [Slices] (slave only)	Max. Size [Slices] (full master-slave)
PLB v3 64b	180	2593
PLB v4 64b	160	1764

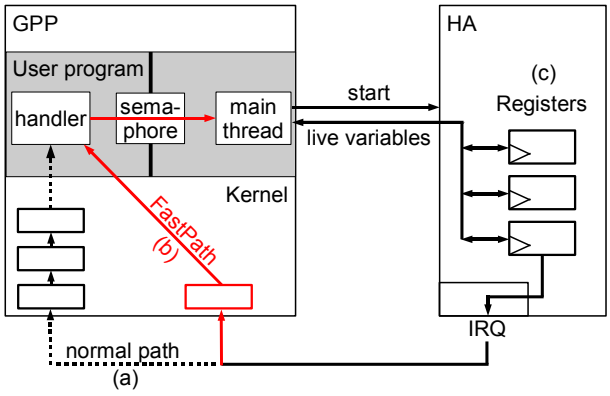

Fig. 8. FastPath: Low-latency SW calls and live variable transfers

In addition, the bus wrappers needed to attach an HA to a PLB-based system require significant chip area (see Table 3) and also add latency (due to the complexity of the PLB protocol). Combined with the limited burst length of just 16 words, these misfeatures render the memory subsystem incapable of actually achieving the 64 bit, DDR-200 operation that should yield the peak performance of 1600 MB/s on the ML310 board (discussed in greater detail in [36]). The new point-to-point interconnect used on the Virtex-4 FX and -5 FX devices does not address this problem either, since it does not allow the direct connection of an HA to memory.

6 OS Integration: Low-Latency GPP↔HA Communication

As described in Section 2, our execution model relies on the ability to quickly switch execution between GPP and HA processing to avoid expending reconfigurable area for infrequent or only inefficiently synthesizable operations.

Our **LOWLAT** requirement states that such switches should be performed with minimum overhead. When running the embedded system only with a lightweight RTOS (or even without an OS at all), this is easily achievable. However, with the trend to use full-scale OSs (such as Linux or Windows) even in embedded applications, it becomes increasingly difficult, since task switches and interrupt processing (required for our HW/SW switches) take much longer.

One of our contributions consists of HW/SW mechanisms to achieve **LOWLAT** even in such hostile environments. As a baseline, consider that even with a Linux version patched for low-latency execution, the interrupt response latency on the ML310 platform is 62μs. This is due to the numerous SW layers that an interrupt has to pass through before it reaches the actual application-level handler in user SW (e.g., to perform a memory allocation requested by the HA).

This structure is shown in Figure 8a: The *top part* of the standard IRQ handler acknowledges the interrupt request, creates a tasklet, also known as *bottom part*, and immediately exits. The actual execution of the bottom part is deferred subject to the normal Linux scheduling rules. When it is scheduled, the bottom part dequeues the suspended process which is waiting for the interrupt (e.g., using `acs_wait()` in Figure 13) from the wait queue. But again, execution of the process is deferred until it is picked by the scheduler. Only then can the application-specific handling (e.g., the memory allocation) actually be performed. The high overheads associated with this procedure permit HA/SW switches only at a very coarse granularity (= long time intervals between switches), otherwise the total interrupt processing overhead becomes excessive.

We have taken two measures to reduce the latency of such an exchange: On the HW side, we now employ the *dedicated* formerly unused *critical interrupt* vector of the PowerPC 405 for HA ↔ GPP signaling, which has a higher priority than the conventionally used *external interrupt* (but note that we have taken explicit measures to prevent priority inversion). On the SW side, incoming HA interrupts (which request SW Services) are now processed by *directly branching* to the application-level handler in user space (a C-callback function, shown in Fig. 8b) without the two traditional scheduler interventions. Virtual addressing and access permissions are set up to allow the handler code full access to user space data (including global variables and library functions). Synchronization between the application's main thread and the callback handler is achieved (as usual) via a semaphore. However, in contrast to the comparatively heavy-weight semaphore facilities offered by the C library, we realize the semaphore based on the formerly unused special GPP register USPRG0. Employing a special processor register avoids the bus contention and memory accesses which are associated with conventional in-memory semaphores, but does not increase pressure on the compiler's general-purpose register allocator.

The main application thread continuously polls the semaphore (again, a cheap operation due to it being a GPP register) to retrieve which SW Service is currently required by the HA. Such task switch-avoiding polling is becoming increasingly common in many modern device drivers [16], [17].

At first glance, the weakness of this scheme appears to be that the main application thread continues to execute under control of the Linux scheduler, thus eliminating the advantages of the quick semaphore operation. While an instinctive response would be to increase the scheduling priority of the main thread, such steps should only rarely be necessary in practice: If operations are so latency-sensitive that they would suffer even from at most one scheduling round (which might occur during the main thread if the HA execution period exceeds the allotted time slice of the process), they can be moved to the callback handler itself and thus be executed immediately on receiving an interrupt. Remember that code in the callback has the same capabilities as in the main thread. We term these combined measures for achieving our **LOWLAT** objective *FastPath* and will show in Section 9 how they lead to significant reductions in interrupt latency. They support the fast HA/SW switches as required by our execution model and allow a finer GPP/HA

```
...
// get pointer to HA registers
ha = acs_get_ha_regs(NULL);

printf("ha[0] Value after RESET = %x\n", ha[0]);

// write new value
ha[0] = 0x87654321;
printf("ha[0] New value = %x\n", ha[0]);
```

Fig. 9. Fast Variable Exchange

partitioning granularity. Interestingly, the reduced interrupt response time has proven to be nearly independent of the system load in our experiments.

In addition to quick GPP/HA signaling, **LOWLAT** also requires the low-latency exchange between GPP variables and HA registers (live variables). When memory-mapping the HA registers, the GPP can read a value in 20 ns and write it in 40 ns. Since SW Services generally require only very few variables [40], the transfer time is negligible relative to the signaling latency (which takes several microseconds, see Section 9). Figure 9 shows sample code for such a live variable transfer.

7 High-Performance Memory Access

LOWLAT is only one requirement for an efficient adaptive computer. Section 5.2 already discussed that the vendor-recommended rSoC architecture was not able to fully support the bandwidth of the physical memory chips. To actually achieve our **HAMEM** objective, we now exploit the reconfigurable nature of the rSoC to implement alternative approaches.

As with FastPath above, we aim to replace generic, but potentially complex (and slow) structures with specialized, but much faster ones. The main idea of our *FastLane* high-performance memory interface [36] is the *direct* connection of the memory-intensive HA to the central memory controller *without* an intervening PLB (shown in Figure 10, please disregard the optional MARC interface, explained later in Section 7.1, for the purposes of this introductory discussion). This approach eliminates two levels of wrapper latency and area (see Table 3) between HA and memory controller as well as the need for bus arbitration. Furthermore, it allows the use of a light-weight double-handshake protocol instead of the more complex PLB one. Full speed 64 bit double-data rate memory access is now available to the HA. Also, the PLB-side wrapper of the memory controller, which is still required to provide the GPP with memory access, can now *forward* GPP requests to the HA (for the slave-mode live variable exchange under GPP control). This sharing of a wrapper reduces area and the load on the PLB signals, which improves system-wide timing.

The original version of FastLane [36] has since been improved by doubling its data path width to 128 bits (still running at 100 MHz on the ML310). The current implementation, termed FastLane+, has thus access to the full bandwidth of the memory controller.

7.1 Abstracting Memory Interfaces

During our research in reconfigurable computing, we have found it extremely useful to introduce higher-level abstractions

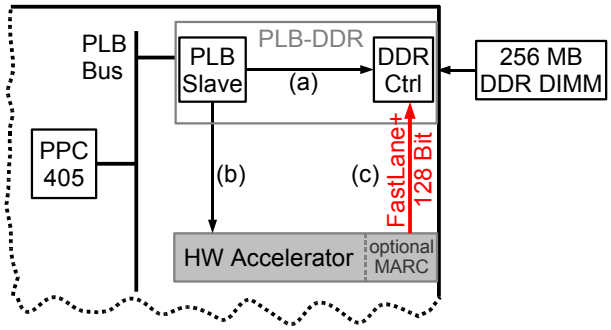

Fig. 10. FastLane+: Attaching HA directly to DDR controller

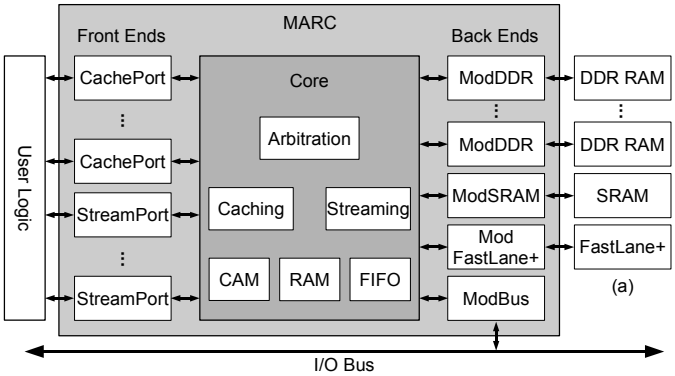

Fig. 11. MARC overview

into our system architectures. One such abstraction is the Memory Architecture for Reconfigurable Computers (MARC, shown in Figure 11). Instead of being focused on physical memory characteristics, as a classic memory controller would be, MARC deals with the *semantics* of memory accesses. Specifically, it provides separate ports for regular streaming accesses and irregular cached accesses. Our HW/SW compiler COMRADE uses MARC as a general memory abstraction, thus enabling the generation of HA cores which are portable between different ACS systems.

MARC, which is described in greater detail in [39], consists of three main parts:

- The *core* encapsulates the functionality for caching and streaming services, the cache tag CAM (Content Addressable Memory), cache line RAM and stream FIFOs. The core also arbitrates the *back ends* and *front ends*, aiming to keep all of them working concurrently but resolving conflicts when accessing the same resource.
- The *front ends* provide standardized, simple interface ports for both streaming and caching using a simple double-handshake protocol.
- The *back ends* adapt the core to several memory and bus technologies. New back ends can be easily added as required.

We have extended MARC with a FastLane+ back-end (Figure 11a) to allow both our library of manually designed as well as automatically compiled HAs to seamlessly benefit from the new memory system (see Figure 10, now *including* the optional MARC interface).

7.2 System Architecture Issues

While a more efficient memory attachment of the HA is an enabler for improved performance, the issue of HA-memory integration must also be considered at the system level. The GPP, under control of the OS, continues to require its own access to main memory and may be intolerant of longer access delays when the HA uses main memory for extended periods of time. Note that even on an otherwise idle system, OS services such as interrupt handling, timers, and process scheduling cause memory traffic. Bus master I/O devices such as network interfaces that send/receive data directly from/to memory would be affected similarly if the HA hogged access to the memory bus. In both cases, memory responses that are significantly delayed by HA activity may lead to instability at the system level (lost packets, imprecise timers, etc.).

Thus, we have to give the GPP and bus master devices *priority* over the HA (since the latter can be explicitly designed to tolerate memory access delays). The arbitration logic required for this behavior is implemented within FastLane+. It ensures that the GPP and bus master devices may interrupt HA accesses, but that the HA always has to wait for an idle memory bus before proceeding. These rules enforce OS scheduler decisions at the HW level (**OSSCHED**): The HA can never hinder a SW process scheduled for execution by starving it from memory.

FastLane+ thus switches between *active* and *passive* modes. In passive mode, GPP memory requests which are received via PLB are transparently passed to the memory controller (Figure 10a), while the HA cannot access main memory. It can, however, reply in slave mode to GPP requests to read/write memory-mapped HA registers (Figure 10b, also see Section 6). Such a slave access can occur in parallel to a memory transfer, e.g., by a bus master device. In active mode, the HA is granted access to main memory (Figure 10c). Should the GPP request memory during that time, that request is briefly delayed until the currently active HA memory transaction can safely be terminated (obeying memory controller access protocols). Note that the reverse is not true: In passive mode, the HA can never interrupt a GPP access to memory.

8 OS Integration: Virtual Memory

While FastLane+ allows the highly efficient sharing of physical memory between the HA and the rest of the system (GPP, bus master devices), this issue also has to be considered further at the OS level, specifically in terms of both efficient and secure integration.

Integrating a fully master-mode capable HA into an OS that supports virtual memory (VM) is not trivial. The GPP relies on a dedicated MMU (see Figure 12) to translate the virtual *user space* addresses which are used in SW applications into physical addresses suitable for sending out on the PLB. The HA, however, traditionally cannot access the GPP-integrated MMU and *physically* addresses memory. In general, this implies that HA/SW communication in a VM setting has to use *both* virtual and physical addresses. Furthermore, the GPP has to ensure that all data is physically present in memory before the HA is started, since the HA usually can neither initiate nor handle page faults. The next sections describe three increasingly capable approaches for HA/SW interaction in a VM environment.

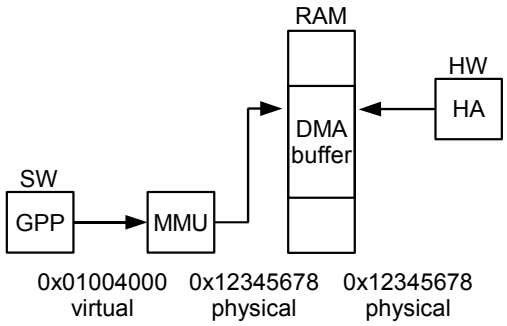

Fig. 12. Hardware and software addressing of memory

8.1 Initial Approach: In-Memory DMA Buffer

In Linux, a *direct memory access buffer* (DMA Buffer) is the simplest approach to the problem. It consists of a contiguous range of pages locked into physical memory. As described above, a DMA Buffer requires the use of both physical (in the HA) and virtual addresses (on the GPP) for the same memory locations.

In Figure 12, a SW program has allocated a DMA Buffer and passes its physical address to the HA. The SW accesses data in the Buffer relative to its user space base address `0x01004000`, which the MMU translates to the physical address `0x12345678`. The HA directly uses this physical address to access data in the DMA Buffer.

A sample program using this approach is shown in Figure 13. Two new DMA Buffers (`..._in` for input to the HA, `..._out` for data output from the HA) are allocated by calls to `acs_malloc_master()`, which returns both addresses for each Buffer. SW then reads the input data from a file into the buffer, accessing it via its virtual user space address `buffer_in`). The physical addresses of both input and output Buffers are then passed to the HA, which is now started. It uses physical addresses to read and write the data from/to its respective Buffer. Once it finishes execution, SW reads back the results from the output buffer, once more using the virtual address, and saves them to a file.

8.1.1 Limitations

DMA Buffers on their own provide basic HA/GPP interoperability in a VM OS-compatible fashion. But this usage has a number of disadvantages with respect to our requirements on ACS architectures (Table 1). Since the HA and the GPP use *different* addresses (physical and virtual), it is impossible to share pointer-based data structures (very common in C programs) between HW and SW processing. Such a restriction violates our **ADDRESS** requirement. While the general approach is easy to implement, using two addresses for the same data will be unfamiliar to many SW developers. Furthermore, initialized data kept in a program's static `.data` or `.bss` segments, as well as variables dynamically allocated on heap and stack have to be explicitly copied to and from the DMA Buffers if they are to be processed by the HA. These copy operations, which

```
...
// get virtual address pointer to HA registers
ha = acs_get_ha_regs(NULL);

// request memory for input and output arrays
// retrieves both virtual and physical addresses
buffer_in =
  acs_malloc_master(NUM_WORDS*sizeof(*buffer_in),
                    (void **) &buffer_phys_in);
buffer_out =
  acs_malloc_master(NUM_WORDS*sizeof(*buffer_out),
                    (void **) &buffer_phys_out);

// fill buffer from file using virtual address
fread(buffer_in, sizeof(*buffer_in), 1, file_in);

// transfer physical addresses to HA
ha[REG_SOURCE_ADDR] = buffer_phys_in;
ha[REG_DEST_ADDR]   = buffer_phys_out;
// number of data words
ha[REG_COUNT]       = NUM_WORDS;
// start command
ha[REG_START]       = 1;

// wait for end of computation (IRQ)
acs_wait();

// write buffer to file using virtual address
fwrite(buffer_out, sizeof(*buffer_out), 1, file_out);
...
```

Fig. 13. API example for DMA Buffer

are required before and after every HA execution, can take a significant amount of time (see Section 9.3) and would violate our requirement **HAMEM**.

Linux DMA Buffers are also generally considered as *non-cacheable* memory areas. This reflects their normal (non-ACS) use for communicating with bus-master I/O devices. Since these devices can write to the Buffer without the GPP being aware of it, disabling caching prevents the GPP from reading stale cached data instead of the current Buffer contents. While this approach performs acceptably with the usual limited Buffer sizes ($\approx$ 64KB), it is inappropriate in an ACS scenario that potentially needs to share much larger data structures between GPP and HA. If we were to mark such a large block non-cacheable, all GPP accesses to it would be significantly slowed, violating our **SWPERF** requirement.

8.2 Refined Solution: AISLE

The Accelerator-Integrating Shared Layout for Executables (AISLE) is a refinement of the basic DMA Buffer technique. We now keep all data areas of a SW executable (stack, heap and data segments) *inside* of a larger DMA Buffer at runtime, thus eliminating the time-consuming copy operations [37]. This also has the effect of making pointers *freely interchangeable* between HA and SW, which fulfills our **ADDRESS** requirement. The latter relies on the observation that the virtual and physical addresses of *all* data memory locations now differ only by a *constant* offset. This offset can be transparently corrected within the HA address generation unit, which enables the application on the HA to operate on the *same* virtual addresses as user space SW on the GPP. In this fashion, data is now *implicitly* shared between HA and GPP as required by the currently executed algorithm, without need for explicit copies or pointer relocation operations.

Since AISLE changes the ordering of segments at program load time, we altered the part of the Linux kernel which is responsible for loading executables in the common Executable and Linking Format (ELF) [24] from disk into memory.

The normal in-memory layout of executables is shown in Figure 14a. Note that due to the demand-paging which is possible in a VM OS, individual pages of a segment are read from disk to memory only when they are actually required by the program (in the form of a page fault). The main task of the original ELF loader is to establish a mapping between the underlying disk data and virtual memory addresses: Instructions in the .text segment as well as various forms of data segments (.data etc.) are mapped starting at virtual address 0x10000000. Memory areas that are managed at run-time and have no underlying data in the executable file (such as the stack and heap) are mapped to so-called *anonymous memory*: This will happen for the heap, growing upward from the end of the .bss segment, and the stack, growing downward from virtual address 0x80000000.

Our AISLE arrangement is shown in Figure 14b. At its heart lies a DMA Buffer, which is allocated at a known virtual address with a fixed size (16 MB in the Figure, encompassing addresses 0x10000000 to 0x10FFFFFF). This allocation occurs when the first AISLE executable is to be loaded. AISLE executables are standard ELF executables that have a specific flag bit set in the header and which were created using a linker script that moves all executable code *outside* of the DMA Buffer at link time. Since the HA can only access the DMA Buffer, the executable code of the program is protected from a rogue HA, achieving our requirement **PROTCODE**. This also exploits the limited DMA Buffer space more efficiently (just for data storage) and potentially allows the use of narrower address buses within the HA.

An AISLE executable is no longer completely demand paged. Instead, in the Linux kernel function `do_mmap_pgoff()`, our modified ELF loader now directly reads data from the executable file into the DMA Buffer. Since the HA does not have access to the GPP MMU, and thus cannot cause page faults which would demand-page accessed data from disk to memory, we have to assure that *all* data of the program is present in physical memory before HA execution (but see Section 8.3 for an alternative approach). Note that executable code in the .text segment continues to be demand-paged in the usual fashion even for AISLE executables.

Furthermore, we also have to ensure that dynamic memory will only be allocated in the HA-accessible DMA Buffer. To this end, we modified the kernel function `do_brk()`, which is responsible for extending the virtual address space of a process, to hand out memory located in the DMA Buffer instead of HA-inaccessible anonymous memory. Analogously, we moved the downward-growing stack into the DMA Buffer by altering `setup_arg_pages()`. Note that all of these modifications only become active if the AISLE flag is set in the ELF header, conventional ELF executables are fully demand-paged as usual.

HA-internal addresses are clamped by FastLane+ (cf. Section 7) to lie only within the DMA Buffer. We thus not only achieve **PROTCODE** for the SW process that controls the HA, but protect the entire rest of the system (code *and* data) from a

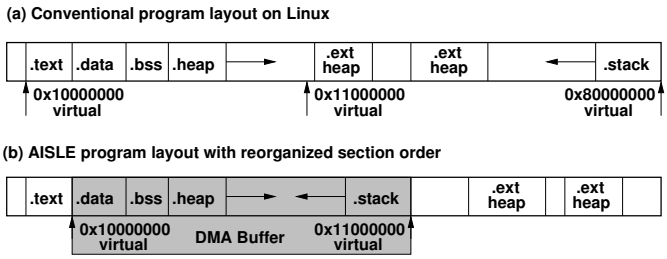

Fig. 14. Conventional and HA-compatible program layouts

potentially erroneous HA (**PROTSYS**).

Giving the HA the **HAMEM** ability induces a heterogeneous multi-processor architecture for our ACS. Cache synchronization in such a setting often relies on bus snooping logic or special bus protocols (MESI, MOESI, etc.). However, such capabilities are rarely (if ever) present in embedded systems GPPs. To apply our ACS execution models even in such restricted architectures, we rely on SW for coherency between the GPP cache and the (possibly HA-modified) DMA Buffer.

When starting the HA via our control API, we first invalidate and flush to memory all dirty GPP cache lines that hold addresses located in the DMA Buffer. Clean lines located in the Buffer are just invalidated, lines outside of the Buffer are not affected at all. When SW execution resumes on the GPP (HA execution is completed or suspended for a SW Service), all GPP accesses to the Buffer retrieve fresh data. Thus, we can now make the AISLE DMA Buffer cacheable and allow the SW process to run at full speed (the **SWPERF** requirement).

Our AISLE-enhanced Linux fulfills all of the requirements demanded by the COMRADE execution model in a manner completely transparent to SW developers: C programs need neither explicit copy operations, nor specialized management functions for HA-accessible memory. An example of this is shown in Figure 15. Memory is allocated by just declaring variables or using familiar standard C library functions. Only a single kind of address (virtual) is used in the combined HA/GPP application.

We have implemented additional ways to customize this functionality. For example, an HA-accelerated program might require large I/O buffers that need not be HA-accessible. When using AISLE in default mode, these buffers would also be allocated from DMA Buffer space (along with the rest of the program's data). However, increasing the DMA Buffer size comes at a cost: First, more memory is removed from the OS demand-paged physical memory pool. Second, the HA needs wider address buses to access the larger DMA Buffer. Large data structures that do not actually require HA accessibility thus waste precious DMA Buffer space. To also support these use-cases efficiently, *optional* calls in our SW API can explicitly request HA-inaccessible memory from the extended heap (shown as .ext heap in Figure 14). The functionality is provided by the standard C library function `mallopt()`: It can set a threshold on allocation sizes, that, when exceeded, causes an expansion of the entire heap. This expansion uses the `mmap()` call that will add new virtual memory *outside* of the normal HA-accessible area to the heap. Known SW-only data is thus

```
...
// get virtual address pointer to HA registers
ha = acs_get_ha_regs(NULL);

// memory for input and output arrays
// in standard program data area
int buffer_in [NUM_WORDS];
int buffer_out[NUM_WORDS];

// fill buffer from file using virtual address
fread(buffer_in, sizeof(*buffer_in), 1, file_in);

// transfer to HA same virtual addresses shared with GPP
ha[REG_SOURCE_ADDR] = buffer_in;
ha[REG_DEST_ADDR]   = buffer_out;
// number of data words
ha[REG_COUNT]       = NUM_WORDS;
// start command
ha[REG_START]       = 1;

// wait for end of computation (IRQ)
acs_wait();

// write buffer to file using virtual address
fwrite(buffer_out, sizeof(*buffer_out), 1, file_out);
...
```

Fig. 15. API example for AISLE

prevented from bloating the AISLE DMA Buffer.

8.2.1 Limitations

AISLE fulfills our requirements, but does have limitations. One of these is the mostly static size of the underlying DMA Buffer: It is allocated with a fixed size when starting an AISLE program (generally the maximum size of static and anticipated dynamic data). While resizing the Buffer at run-time is possible, this is a relatively slow operation. Also, as already described above, the Buffer physical memory is no longer available for demand paging. This can slow down program loading (since data areas are always completely loaded to memory instead of being demand-paged) as well as the entire system (reduced physical memory can lead to more paging).

8.3 Full Virtual Memory Support in the HA: PHASE/V

In contrast to the light-weight, but limited AISLE technique, our Processor Hardware-Accelerator Shared Environment with Virtual Addressing (PHASE/V) [38] promotes the HA to a fully capable VM participant. PHASE/V shares the complete virtual address space between GPP and HA (instead of just a DMA Buffer as in AISLE). All VM features, such as demand paging, swap space, copy-on-write, and file-backed `mmap()` mappings, are now supported for both HW and SW applications. Both AISLE limitations (slower program loading, reduced physical memory for paging) are lifted with PHASE/V.

These enhanced capabilities are accompanied by a significantly increased implementation complexity. The PowerPC 405 which is employed as GPP in our ACS does not allow external access to its MMU (Figure 16) and is thus typical for most embedded processors. Hence, we implemented a separate MMU in the HA to allow it VM operations independently of the GPP MMU. This HA-MMU manages a TLB of 64 direct-mapped entries which is implemented in a carefully tuned manner for our Virtex II Pro FPGA: Lookup-tables (LUTs) in RAM mode are used to compose the memories holding the

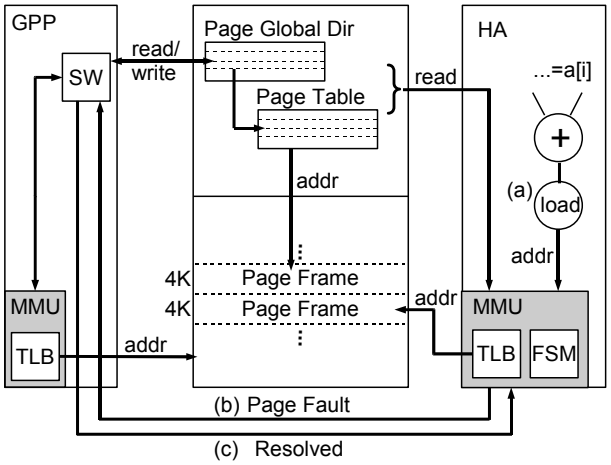

Fig. 16. PHASE/V MMU system: HA↔OS interactions

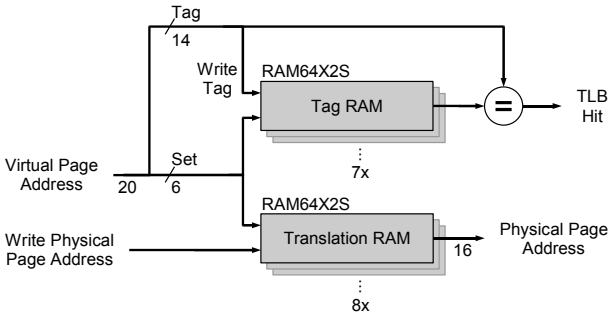

Fig. 17. PHASE/V MMU system: Tag- and translation RAM

physical address translations and tags. Such LUT-RAMs have 64 entries, each 1…2 bits wide, which can be read in a single cycle. Since we use 4 KB pages (for compatibility with Linux), a 32 bit virtual address has a tag component of 14 bits, 6 bits of direct-mapped TLB index, and 12 bits page offset. We employ seven 2-bit RAMs with 64 entries each (RAM64X2S in Figure 17) to hold the tags. The physical page address would require 20 bits to cover the entire 32 bit address space (due to the 4 KB pages). However, since the stock ML310 board that we use as ACS only has a total of 256 MB physical memory, 16 bits of physical page address suffice in practice. These fit in eight RAM64X2S blocks (leading to the total of 15 blocks shown in the Figure). The tag comparator itself is realized using the fast carry-chain logic that is present in the Virtex II Pro CLBs.

When the HA initiates a memory operation using a virtual address (e.g., a load node in the HA data path, see Fig. 16a), a tag lookup is performed. On a hit, the virtual-physical mapping is present as an entry in the physical address translation LUT-RAM, and delivered to main memory (via FastLane+) within a single cycle (achieving **HAMEM**). On an HA-TLB miss, a walk of the page tables in main memory is required. We achieve this by implementing a small controller Finite State Machine (FSM, cf. Fig. 16) that can evaluate the OS kernel-maintained data structures (Page Global Directory and Page Tables) to determine the correct mapping and cache it in the HA-TLB for later re-use.

Beyond virtual-physical address translation, we also need to handle page faults (if no physical memory page is mapped to a virtual address). This case will be detected when a page table walk performed by the HA-MMU cannot determine a valid mapping. Since handling page faults can get very complex (e.g., fetching data from disk or over a network), we rely on the standard OS mechanisms and just initiate the process. This is done by our FastPath signaling scheme (Section 6), which conceptually treats the page fault as just another SW Service (shown as signal Page Fault in Fig. 16b). It is the responsibility of the Linux kernel to fetch the missing page frame and update the page tables (signal read/write). As usual for a SW Service, execution switches back to the HA afterwards (signal Resolved, Fig. 16c).

Handling the current page fault can cause the eviction of pages that are already resident in physical memory. In this case, they also have to be removed from the HA-TLB if they have a valid entry there. This is achieved by modifying the Linux kernel function `flush_tlb_page()`, which is the only function involved with page flushing / swapping in low-memory situations, to update both GPP- and HA-MMUs.

GPP-HA cache coherency in PHASE/V is handled in SW identically to AISLE. Finally, PHASE/V can support multiple HAs, either sharing the same HA-MMU, or using HA-exclusive ones (this would require additional inter-HA-MMU synchronization mechanisms, though).

8.3.1 Limitations

While PHASE/V is much more capable than AISLE, it is also more costly: It requires more HW (≈ 300 FPGA slices, see Table 4) and is also susceptible to the same performance risks as a GPP MMU: When an application's (SW or HW) working set of virtual addresses exceeds the TLB (GPP or HA) capacity, the resulting TLB misses cause frequent page table walks (also known as *TLB thrashing*) which can lead to severe application slow-downs (this will be examined in Section 9.3). To some extent, this could be countered by larger or fully associative TLBs. However, in addition to requiring even more area, both of these approaches generally have slower implementations (violating our current single cycle translation at 100 MHz).

9 Evaluation

Each component of the different solutions that we have discussed is now evaluated separately to better understand its respective impact on the overall system performance. We first show the advantages of our FastPath quick signaling solution over standard OS IRQ handling and then give results for the FastLane+ enhanced memory system. Finally, we compare the techniques for enabling HA-operation in a VM environment.

9.1 FastPath

The main contribution of FastPath is a reduced HA/SW signaling latency. Since a common aim of using an ACS in the first place (instead of a sole GPP) is improved performance, we will examine the impact of signaling latency (communication overhead $t_{overhead}$) on the effectively achievable speed-up.

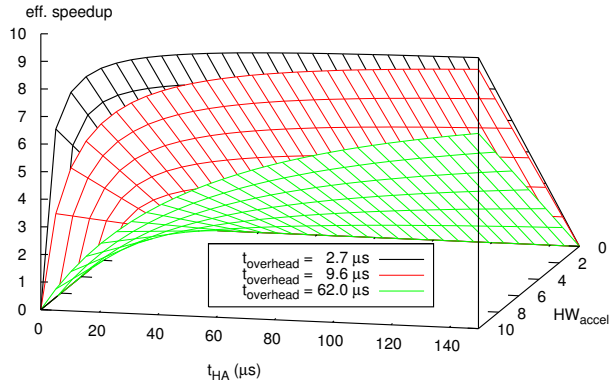

Fig. 18. Effective speedup as function of HA execution time and raw HW acceleration factor for different latencies

We compute the *raw GPP-to-HA acceleration factor* as $\mathrm{HW}_{accel} = t_{SW}/t_{HA}$, the ratio of times for running an algorithm (just part of a program) in SW to that of performing the same computation on an HA. Note that this raw factor does not consider the communication overhead yet. However, the *effective speed-up factor* for that part of the program also includes the overhead:

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

The *communication overhead* is computed as $t_{overhead} = t_{IRQ} + t_{sem}$. We define t_{IRQ} as the time interval 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). t_{sem} is the time between the handler setting the semaphore and the resumption of the main program thread.

We measured these times for different scenarios using dedicated cycle-accurate HW counters. The standard Linux interrupt path (employing the usual tasklet-driven wait queue in lieu of our fast semaphore) on the ML310 ACS has $t_{overhead} = 62\mu s$. 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 6), we can thus shorten the total overhead by a factor of 6.5x to 23x over the standard case (achieving **LOWLAT**).

If a serialized single-threaded execution model (the default Linux case) is not required, the IRQ-handling callback function (which has full access to virtually addressed user space data and system libraries) can directly perform latency-critical operations without synchronizing to the main thread (thus creating a dedicated callback thread), which would improve $t_{overhead}$ by another 9…22% even over our fast register-based semaphore.

Looking further, FastPath running on our relatively slow 300 MHz embedded GPP outperforms even specialized real-time variants of Linux, such as RTAI/LXRT, or carefully tuned versions of recent Linux 2.6 kernels running on multi-GHz desktop CPUs [18].

Figure 18 shows the system-level effects of the different values for $t_{overhead}$. It graphs the achievable effective speed-up (z-axis) for a given raw HA speed-up (HW_{accel}, y-axis) and time spent in the HA before switching back to SW execution (t_{HA}, x-axis).

TABLE 4
FPGA areas for rSoCs with specified HA at 100 MHz clock

HA Type	Slices	4-LUTs	Flip-Flops	BlockRAM
Copy-V2P-Ref	8408	9868	7596	28
Copy-FastLane+	6952	7904	6408	81
List-DMA	6660	7450	6001	24
List-AISLE	6664	7451	6009	24
List-PHASE/V	6898	7731	6146	24

The bottom surface shows the effective speed-ups that are achievable when using the standard Linux signaling mechanism, the two upper surfaces show the impact of FastPath (top: best-case latency, middle: worst-case latency). The impact of an HA on total performance obviously increases when the HA is used for longer periods of time t_{HA} and it can actually leverage the raw speedup itself. Shorter values for t_{HA}, either due to smaller algorithms being offloaded to the HA, or due to the HA requesting SW Services, reduce the effective speed-up. For very short HA execution times, the relatively high signaling overheads turn even high raw speed-ups into small effective speed-ups (or even slow-downs, effective speed-up < 1).

More interesting is the behavior for intermediate values of t_{HA}. E.g., the two surfaces representing FastPath show that 90% of the raw speed-up is effectively achievable after just $23\ldots75\mu s$ of HA execution time, while the conventional signaling requires much longer HA execution times to achieve similar speed-ups.

With FastPath, HW/SW partitioning can be performed at much finer granularity than using the conventional scheme: Even smaller kernels that would not yield actual speed-ups (due to the communication overhead) can now effectively be accelerated in HW.

9.2 FastLane+

When evaluating the system-level performance impact of FastLane+, we consider the SW and HW sides separately: To stress-test the memory system and buses, we use an HA which repeatedly copies a 2 MB buffer from one memory location to another as quickly as possible, yielding 4 MB of reads and writes per turn. The area statistics of the vendor reference and FastLane+ implementations of the rSoC containing this Copy-HA are given in Table 4. On the SW side, we run a number of programs (independently of the Copy-HA) which we carefully selected for their specific load characteristics (described below). We then measure both the SW and HA execution times for different combinations, which include the extreme cases of either the HA or the GPP being idle.

Our test suite of programs (described in greater detail in [37]) contains both desktop and embedded applications:

- scp provides a mix of CPU- and I/O load [30]
- netcat exercises network I/O exclusively [31]
- gcc interleaves short I/O- and long calculation phases [32]
- GSM provides codec stream data processing [33]
- imgpipe implements multi-stage image processing [34]

Our first test scenario measures the memory performance which is achieved by the Copy-HA while the given SW program

TABLE 5
Copy-HA runtimes and available throughput using original and FastLane+ memory subsystem implementations under various GPP software loads

GPP SW Load	V2P ref design Exec Time [ms]	V2P ref design Mem Rate [MB/s]	FastLane+ Exec Time [ms]	FastLane+ Mem Rate [MB/s]
idle sys	18.81	213	5.62	1424
scp	55.11	73	12.61	634
netcat	53.07	75	19.51	410
gcc	32.14	124	17.98	445
GSM	19.05	210	6.03	1326
imgpipe	44.67	90	19.37	413

TABLE 6
Software run times and slow-down on idle system and using Copy-HA attached by original and FastLane+ memory subsystem implementations

GPP SW Load	HA inactive [ms]	V2P ref design [ms]	V2P ref design slow	FastLane+ [ms]	FastLane+ slow
scp	4831	61052	1263%	5788	120%
netcat	3130	55938	1787%	3843	122%
gcc	40686	166655	409%	52526	129%
GSM	25981	40045	154%	27357	105%
imgpipe	3545	5109	144%	3806	107%

is executing on the GPP. In Table 5, we show both the HA execution time to perform the copy as well as the corresponding memory throughput. The HA is integrated with the rSoC first by the vendor-provided Virtex II Pro reference design, and then using our FastLane+ architecture. The advantages of our approach are obvious: FastLane+ increases the throughput significantly under all SW load scenarios, in some cases by a factor of 8.6x. This considerably improves the **HAMEM** requirement of our execution model. With only Linux (but no other user applications) running, we can make 89% of the theoretical physical memory throughput available to the HA.

Next, we evaluate the effect of running the HA at full speed on the execution times of the SW applications on the GPP. We give three run-times for each application, corresponding to the Copy-HA being inactive (not performing transfers), and the two ways of attaching the Copy-HA to the system. The column 'slow' shows the increase in SW execution time when the Copy-HA is active (e.g., a value of 107% here indicates that the application is 7% slower than with an inactive Copy-HA).

The use of the vendor-provided reference design leads to significant slow-downs (up to 17.9x) of SW programs once the Copy-HA becomes active. With FastLane+, though, all SW programs are significantly less slowed, despite the high memory throughput that is achieved by the HA copying data in the background. Our technique to accomplish **OSSCHED** by giving the GPP absolute priority over the HA with regard to memory access is thus shown to be successful.

The differences in slow-down for FastLane+ are due to the different load characteristics: gcc is slowed most, since it does only little I/O but spends much time transforming in-memory data structures (preempting the Copy-HA). scp and netcat perform more I/O and fewer memory accesses (less interference with Copy-HA), and are thus slowed less. gsm and imgpipe run mostly out of the GPP caches and execute almost without interference from the Copy-HA. Note that the reference design has a different slow-down profile: The HA and GPP share not just main memory, but also the PLB which arbitrates between accesses to memory and I/O devices. Thus, the more I/O intensive applications in the reference design are slowed down further by the Copy-HA than with FastLane+.

If even higher memory throughput is required and interrupt processing can be stopped temporarily, the GPP could be completely halted (e.g., by freezing its clock signal). In this case, FastLane+ does indeed make 100% of the physical memory bandwidth available to the HA and outperforms the vendor-provided solution again, which just reaches efficiencies of 25% for reading and 33% for writing in this scenario due to the limited PLB burst length and frequent rearbitration.

Since FPGAs are often used in low-power embedded environments, we have also evaluated the power impact of our techniques. However, our measurements show that the active power consumption of our accelerators pales in comparison to that of the PowerPC and the rest of the system (e.g., network interfaces, memories, etc.). The peak difference between system idle power and power consumed with both CPU and accelerator under full load is less than 5%. This does not come as a surprise, since our hardware components encompass at most a few hundred LUTs and Flip-Flops, the entire rest of the SoC remains unchanged. Of course, total *energy* consumption is significantly reduced by our approach since we achieve much shorter hardware and software execution times (see Table 5, compare with Table 6).

9.3 AISLE vs. PHASE/V

AISLE and PHASE/V are evaluated using a HA which exploits the **ADDRESS** capability of our system: SW creates a randomly linked list, with each list element consisting of a 32b integer value and a pointer to the next element. The List-HA traverses the list and performs a dummy operation on the value (if odd, increase by one). We picked such a trivial operation on purpose: This benchmark is designed to test how well the List-HA *accesses* irregular pointer-based data structures (instead of the regular data streaming so common to other ACS applications). We actively avoid the more complex computations that are amenable to greater HA acceleration, since they would unfairly bias the results toward the HA when comparing it to the GPP.

In Table 7, we compare the execution times for different list lengths of a pure SW version of the application with three versions of the List-HA (using a raw DMA Buffer, AISLE, and PHASE/V for memory addressing). The area statistics for the variants are shown in Table 4. In the raw DMA case, the SW has to explicitly copy the list into the DMA Buffer for HA processing. We ensured that both the original and the copy have the same memory alignment to avoid the slow pointer relocation operation that would otherwise be required.

The List-HA using AISLE outperforms the 300 MHz PowerPC 405 GPP by a factor of $\approx$ 2.5x. It is 8x faster than the conventional approach of explicitly copying data to HA-accessible memory. Note again that we are concentrating *solely*

TABLE 7
Runtimes of the pointer-handling application

List length	Software	List-HA using		
		DMA/copy	AISLE	PHASE/V
16K	7.4 ms	27.6 ms	3.4 ms	3.5 ms
32K	15.8 ms	55.1 ms	6.8 ms	12.2 ms
64K	33.0 ms	110.1 ms	13.7 ms	29.5 ms
128K	68.3 ms	220.1 ms	27.4 ms	64.0 ms

on evaluating data access times, the potential for accelerating a more complex *algorithm* by the List-HA is deliberately not considered here.

When comparing AISLE and PHASE/V in Table 7, it becomes obvious that PHASE/V (as all TLB-based VM architectures) is susceptible to thrashing when the working set of virtual addresses becomes too large. This occurs above 16K elements (below, PHASE/V performance matches that of AISLE). In all cases, a List-HA using PHASE/V is still faster than a SW implementation. The designer (or compiler) can thus trade-off between the more efficient AISLE and the full VM participation of PHASE/V on a per-application basis.

10 Conclusion and Future Work

We have introduced a fine-grained model of execution which orchestrates the interaction between conventional GPPs and reconfigurable HAs. Using an actual HW prototype, we demonstrated that, by making suitable architecture choices at the HW as well as OS levels, the fine partitioning granularity which is possible with the model is actually supportable in practice. Thus, even smaller fragments of code can now be moved to the HA with effective speed-ups.

With our improved memory interface FastLane+, we increase throughput by up to 8x and also show that HAs can achieve performance gains not only for conventional stream processing, but also for pointer-chasing applications that were not efficient using prior interfaces (such as PLB). This will hold especially true for algorithms working on large data sets (such as for railway routing graphs [41]) that easily exceed the cache capacities of modern processors, also reducing them to the raw speed of the main memory system. Modern reconfigurable devices *are* already reaching these speeds, but beyond that can then exploit increased parallelism, with regard to both number of memory banks and processing elements.

We have raised the SW abstraction for developing hybrid GPP/HA applications by shielding the programmer from having to explicitly manage main and HA-accessible memory, even in virtual memory settings. Depending on the needs of the application, a developer can choose between the full VM capabilities of PHASE/V and the more restricted, but faster functionality of AISLE.

Note that while we have used a concrete platform to practically validate our approach, our techniques are applicable and generally portable to other target environments.

Increasing the efficiency of PHASE/V is an area of future work: Even using FastPath, the explicit synchronization of the MMUs in the GPP and the HA has a relatively high overhead: The GPP typically requires 210 ns (min. 60 ns, max. 1710 ns) per page table walk to keep its MMU up to date. This could be avoided by sharing a reconfigurable MMU between GPP and HA, and disabling a possible GPP-internal MMU. With the efficiency of our HA-MMU implementation, this would not slow down GPP-MMU operations. Other improvements that are currently under consideration comprise the inclusion of dynamic reconfigurability into execution model and HW architecture as well as the sharing of the reconfigurable area between multiple SW processes.

References

[1] Gokhale M., Graham P.S., "Reconfigurable Computing", Springer, 2005
[2] Synplicity Inc., "Synplify DSP", http://www.synplicity.com/products/synplifydsp/index.html, 2007
[3] Xilinx Inc., "System Generator for DSP", http://www.xilinx.com/ise/optional_prod/system_generator.htm, 2007
[4] Gupta S., Gupta R., et al., "SPARK", Kluwer, 2004
[5] Najjar W., Böhm W, et al., "From Algorithms to Hardware", IEEE Computer, 08/2005
[6] Gokhale M. B., Stone J. M., et al., "Stream-oriented FPGA Computing in the Streams-C High-Level Language", Proc. IEEE Symp. on FCCM, Napa, 2000
[7] Budiu M., Venkatarami G., et al., "Spatial Computation", Proc. Intl. ACM Conf. on ASPLOS, Boston, 2004
[8] Callahan T., Hauser J., Wawrzynek J., "The Garp Architecture and C Compiler", IEEE Computer, 04/2000
[9] MacMillen D., "Nimble Compiler Environment for Agile Hardware", Storming Media LLC (USA), 2001
[10] Kasprzyk N., Koch A., "High-Level-Language Compilation for Reconfigurable Computers", Proc. Intl. Conf. on Reconfigurable Communication-centric SoCs (ReCoSoC), 2005.
[11] Balacco S., "Linux in the Embedded Systems Market (Vol. VII)", Venture Development Corp, 2007
[12] 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
[13] 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
[14] 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
[15] Nallatech, "Intel Xeon FSB FPGA Accelerator Module", http://www.nallatech.com, 2009
[16] Dovrolis C., Thayer B., Ramanathan P., "HIP: hybrid interrupt-polling for the network interface", ACM SIGOPS Operating Systems Review, Volume 35 Issue 4, ACM, 10/2001
[17] Aron M., Druschel P., "Soft timers: efficient microsecond software timer support for network processing", ACM SIGOPS Operating Systems Review, Volume 33 , Issue 5, ACM, 12/1999
[18] Laurich P., "A comparison of hard real-time Linux alternatives", LinuxDevices, 2004
[19] Advanced RISC Machines Ltd., "AMBA AXI Protocol Specification 1.0", 2004
[20] Open Core Protocol International Partnership, http://www.ocpip.org, 2006
[21] IBM Corp., "The CoreConnect Bus Architecture", *White Paper*, 1999
[22] Xilinx Inc., "Embedded System Tools Reference Manual", *Xilinx UG111*, 2006
[23] Xilinx Inc., "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet". *Xilinx DS083*, 2005
[24] Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2, TIS Committee, 1995
[25] Philips Semiconductors, "SAA7146A Multimedia bridge, high performance Scaler and PCI circuit (SPCI)", Product Specification, 2004
[26] Vuletić M., Pozzi L., Ienne P., "Virtual Memory Window for Application-Specific Reconfigurable Coprocessors", Proc. Design Automation Conference (DAC), San Diego, 2004
[27] Garcia P., Compton K.A., "Reconfigurable Hardware Interface for a Modern Computing System", Proc. 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, 2007

[28] Halfhill T.R., “MicroBlaze v7 Gets an MMU”, Microprocessor Report, November 13, 2007
[29] Thatcher T., Hartke P., “OpenSPARC T1 on Xilinx FPGAs - Updates”, RAMP Retreat, Stanford, 08/2008
[30] Friedl M. et al., “OpenSSH 5.2 Release Notes”, http://www.openssh.com/txt/release-5.2, 2009
[31] Giacobbi G., “GNU Netcat 0.7.1”, *user manual* http://netcat.sourceforge.net/download.php, 2004
[32] Stallman R.M., “Using the GNU Compiler Collection for GCC 3.3.6”, http://gcc.gnu.org/onlinedocs/, 2002
[33] ETSI, “Digital cellular telecommunications system (Phase 2+); ANSI-C code for the GSM Enhanced Full Rate (EFR) speech codec”, *standard ETSI EN 300 724 V8.0.1*, 2000
[34] Fisher J., Faraboschi P., Young C., “Embedded Computing: A VLIW Approach to Architecture, Compiler and Tools”, chapter 11.1, Elsevier, 2005
[35] Turley J., “Operating Systems on the Rise”, http://www.embedded.com/columns/showArticle.jhtml?articleID=187203732, 2006
[36] 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
[37] 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
[38] Lange H., Koch A., “Low-Latency High-Bandwidth HW/SW Communication in a Virtual Memory Environment”, Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL), Heidelberg, 2008
[39] H. Lange, A. Koch, “Memory Access Schemes for Configurable Processors”, Proc. Workshop on Field-Programmable Logic and Applications (FPL), Villach, 2000
[40] Kasprzyk, N., “COMRADE – Ein Hochsprachen-Compiler für Adaptive Computersysteme”, Ph.D. thesis, Tech. Univ. Braunschweig, 2005
[41] Müller-Hannemann M., Schnee M., “Finding All Attractive Train Connections by Multi-Criteria Pareto Search”, Proc. 4th Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS), 11/2004

Holger Lange received his diploma in informatics from the Technische Universität Braunschweig (Germany) in 2001. He then joined sciworx GmbH (Germany), a subsidiary of Infineon Technologies, working as a design methodology engineer where he participated in the joint industrial-academic research project “Intellectual Property Qualification”, aiming to develop methods and technology for flexible and automated IP core management. He continued his research activities at the Department of Integrated Circuit Design (E.I.S.) at TU Braunschweig. In 2005, Holger Lange joined the newly founded Embedded Systems and Applications Group in the Department of Computer Science at Technische Universität Darmstadt (Germany), where he started his research on high performance target system architectures for automatic HW/SW compilation. He is currently working toward a doctorate in engineering at the LOEWE Research Center AdRIA, TU Darmstadt. His research interests include reconfigurable computing platforms and efficient embedded systems.

Andreas Koch received his diploma in informatics and his doctorate in engineering in 1992 and 1997, respectively, both from the Technical University Braunschweig (Germany). His doctoral thesis “Regular Datapaths on Field-Programmable Gate Arrays” describes specialized layout-level algorithms for the synthesis of highly-efficient data paths on reconfigurable logic. He then joined the University of California at Berkeley as a Post-Doctoral Scholar, working in the Berkeley Reconfigurable Architectures, Systems and Software Group where he participated in the joint industrial-academic research project “Nimble Compiler Environment for Agile Hardware”, aiming to develop technology for compiling from a high-level programming language to efficient reconfigurable accelerators. He continued this research, both on hardware architectures as well as design tools, after his return to Braunschweig in 1999. It forms the core of his work “Advances in Adaptive Computer Technology”, which was accepted for his habilitation in 2005. Also in 2005, Andreas Koch joined the Technische Universität Darmstadt (Germany) as faculty, heading the newly founded Embedded Systems and Applications Group in the Department of Computer Science. His current research interests include hardware/software co-compilers and associated tools, computer architecture and compute-intense embedded systems. He is a member of ACM, GI, and IEEE.