Seamless Hardware-Software Integration in Reconfigurable Computing Systems

Miljan Vuletić, Laura Pozzi, and Paolo Ienne
Ecole Polytechnique Fédérale de Lausanne

Ideally, reconfigurable-system programmers and designers should code algorithms and write hardware accelerators independently of the underlying platform. To realize this scenario, the authors propose a portable, hardware-agnostic programming paradigm, which delegates platform-specific tasks to a system-level virtualization layer. This layer supports a chosen programming model and hides platform details from users much as general-purpose computers do.

DESPITE THEIR INHERENT POWER and performance drawbacks in comparison with ASICs, FPGAs are increasingly becoming an option for silicon system designers. A way to overcome FPGA shortcomings (such as clock frequencies more than five times slower than those of ASICs and general-purpose processors) is to blend temporal and spatial computing paradigms in systems by using both general-purpose processors and reconfigurable hardware. This is the approach of reconfigurable SoCs (RSoCs) that have recently appeared on the market—for example, Altera Excalibur (http://www.altera.com/literature) and Xilinx Virtex-II Pro (http://www.xilinx.com). Although researchers have reported obtaining significant performance improvements by combining temporal computing (on CPUs) and spatial computing (on FPGAs), two major obstacles hinder the wider acceptance of reconfigurable computing: the lack of a standardized programming paradigm and the lack of portability for codesigned reconfigurable applications.

We propose a general solution that overcomes these obstacles by introducing an additional abstraction. We also address the challenge of achieving seamless hardware-software interfacing and portability with minimal performance penalties.

Programmers should be able to preserve their hardware-agnostic, high-level programming approaches, even in the presence of application parts executed on FPGAs. On the other hand, hardware designers should be able to write accelerators that can run across different platforms, without any change in the hardware description language (HDL) code. To meet these goals, researchers have proposed sequential programming paradigms (represented by user programs with a single execution thread) for reconfigurable computing systems. For example, addressing hardware-software interfacing problems within a compiler considerably improves the programmability of reconfigurable computing platforms.[1] We introduce a more general, parallel programming paradigm (represented by user programs with multiple execution threads), which requires no changes on the compiler side. Recently, researchers have introduced a hardware-centered parallel programming model aimed mainly at supporting the design of networking applications.[2] Other researchers have proposed a hybrid hardware-software architecture that enables a multithreaded programming model by implementing execution support blocks (for example, thread scheduling and synchronization) in reconfigurable hardware.[3]

Our approach is the only one firmly based on the properties that led to general-purpose computing's common acceptance: programming ease and portability. We introduce a multithreaded programming model for reconfigurable computing based on a unified virtual-

0740-7475/05/$20.00 © 2005 IEEE Copublished by the IEEE CS and the IEEE CASS

memory image for both software and hardware application parts. The model supports transparent hardware-software interfacing through an abstraction layer consisting of hardware and software components. A practical implementation of the abstraction layer shows that it offers significant advantages while imposing a limited overhead.

Extending a multithreaded programming paradigm

Our general, parallel-programming paradigm for reconfigurable computing is an extension of the standard multithreaded programming model in which multiple software threads execute in the context of a common process, relying on thread library and OS support for interthread communication and synchronization.[4] We describe the proposed extension with the help of a simple motivational example.

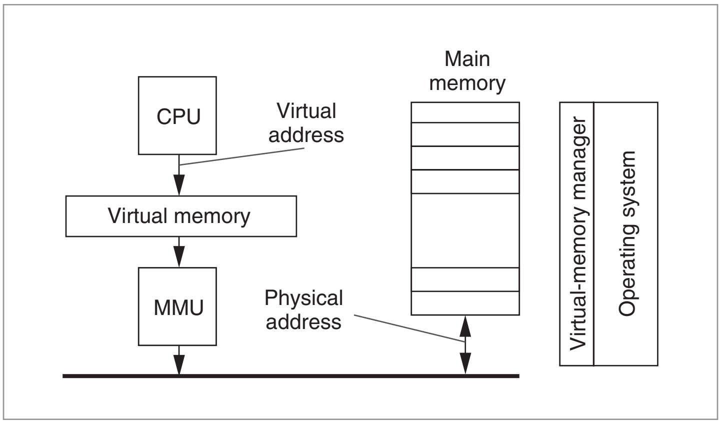


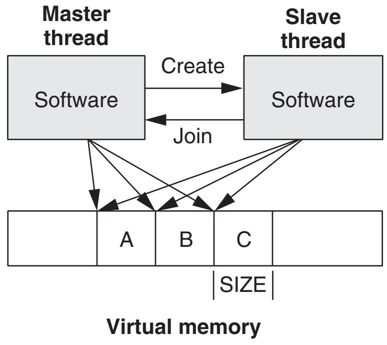
Figure 1. Virtual-memory system. The memory management unit (MMU) translates processor-generated virtual addresses to physical addresses in main memory. The OS part called the virtual-memory manager (VMM) takes care of the translation process.

Standard multithreading

An OS process—a basic OS design concept[5]—represents an executing program with its associated memory address space. Having multiple processes is a costly way to exploit parallelism in applications; using threads is more efficient. Threads execute in the context of a process, so OS bookkeeping costs less, and share the same virtual-memory address space, so data communication is simpler. Virtual memory—a basic computer design concept[6]—simplifies the programming paradigm and allows code portability across systems supporting the same OS but having different memory systems. Figure 1 shows a typical virtual-memory system (without secondary mass storage details).

```
/* Master Thread */
void main() {
        int *A, *B, *C;
        int thr_id;
        …
        read(A, SIZE);
        read(B, SIZE);
        thr_id = thr_create(add_vectors, A, B, C, SIZE);
        do_some_work_meanwhile();
        thr_join(thr_id);
        …
}
/* Slave Thread */
void add_vectors(int *A, int *B, int *C, int SIZE) {
        int i;
        for(i = 0; i < SIZE; i++)
            C[i] = A[i] + B[i];
        }
}
```

(a)



(b)

Figure 2. Vector addition application code (a): After initializing input vectors A and B, the master thread creates a slave thread to compute vector C. Thread creation is similar to a function call, except that the caller and the callee continue execution simultaneously and can work in parallel. The master thread synchronizes with the slave thread using the thread join primitive—that is, it waits until the slave returns. In the application execution, the two threads share the virtual-memory address space and use the same memory pointers (b). Once the computation is finished, the master thread can immediately access the results through its pointer to vector C.

As an example of standard multithreading, Figure 2 shows a simple program that computes the sum of two vectors. The multithreaded application code uses Posix-like thread management[4]—a standard of multithreaded programming. In the master thread, the programmer declares the vector pointers and the slave thread function, initializes the vectors, and creates the slave thread by launching the corresponding function. After doing

```
/* Wrapper Thread */
void add_vectors(int *A, int *B, int *C, int SIZE) {
        int d_chunk = BUF_SIZE / 3;
        int *d_ptr = 0;
        write(HWACC CTRL, INIT); /* initialize accelerator */

        while (d_ptr < SIZE) {
        copy(A + d_ptr, BUF_BASE, d_chunk);
        copy(B + d_ptr, BUF_BASE + d_chunk, d_chunk);
        write(HWACC_CTRL, ADD_VECTORS); /* launch accelerator */
        while () {
            if (read(HWACC_STATUS) == FINISHED) {
                copy(BUF_BASE + 2*d_chunk, C + d_ptr, d_chunk);
                break;
                } else {
                do some work meanwhile();
                }
            }
            d_ptr += d_chunk;
        } …
}
```

(a)

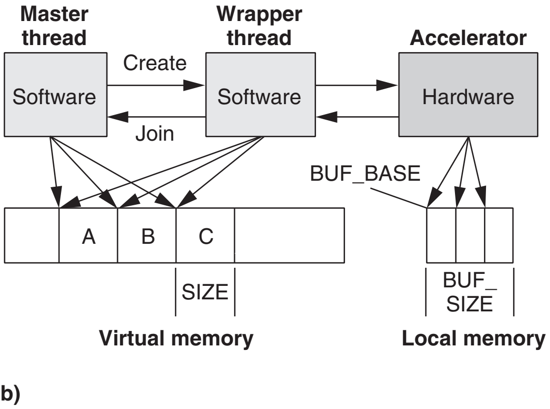


Figure 3. Code for vector addition application in the presence of a hardware accelerator. The master thread is unchanged, but a wrapper thread (a) is now needed to control and transfer data to the hardware accelerator responsible for the computation. The wrapper thread initializes the accelerator, copies data to its local memory, and launches the computation. Input data does not necessarily fit the accelerator's local memory, so the wrapper iteratively copies data back and forth until all data is processed. In application execution, a hardware accelerator and software threads don't share the same memory address space (b). Typically, only partitions of vectors A, B, and C fit in local memory.

some work simultaneously, the master and the slave eventually synchronize through the join primitive.

A key aspect of multithreading is that threads share the same virtual-memory address space. Figure 2b shows the code execution from the memory perspective. The master thread spawns a slave thread that performs vector addition. The slave thread accesses the same memory by using virtual-memory pointers to vectors A, B, and C. Once it finishes the computation, the slave synchronizes with the master thread. The threads share the same memory, while having separate execution stacks.

Including hardware accelerators

Suppose now that in the process of hardware-software partitioning, the designer decides to move the vector addition slave thread to hardware execution. Because no system-level support exists for threads executed in hardware, the programmer must take explicit care of the communication between application software and hardware components. Figure 3 shows a solution for integrating a hardware accelerator, which typically uses a software wrapper thread. The master thread again spawns a slave thread, which is a wrapper to the hardware accelerator. The hardware accelerator accesses a local memory disjoint from the memory address space of the software threads. The size of the data to be processed doesn't necessarily match the local memory's size; therefore, the wrapper thread, and ultimately the programmer, must schedule and perform transfers from main memory to local memory and vice versa.

Figure 3a shows the C code of the wrapper thread (add_vectors). First, three sections of local memory are assigned to partitions of the data for processing, one section for each vector. After initializing the accelerator and entering the main loop, the wrapper thread copies partitions of input vectors A and B to the appropriate sections of local memory. It launches computation and waits for the accelerator to finish, possibly performing some useful work in the meantime. When the accelerator completes the current sections, the wrapper copies back the computed part of result vector C to the user space memory. Pointers are updated and the loop iterates until the accelerator processes all data. Figure 3b shows the code execution from the memory standpoint.

Although the wrapper code is not conceptually difficult to write, it violates encapsulation principles. In fact, if the hardware accelerator didn't visit vector indexes sequentially, we would need to change the wrapper implementation. Furthermore, if we didn't know the access pattern at compile time, we couldn't delegate the vector addition to the accelerator because we couldn't implement the part of the wrapper code that performs copying. As a consequence, programming is burdensome in this inelegant scenario.

Extending multithreading to hardware threads

Our goal is to provide for the seamless integration of hardware and software components in reconfigurable applications. Programmers should not be concerned with interfacing and communication details. They should code their applications as if no differences existed between software threads (executed on CPUs) and hardware threads (executed on FPGAs). For programming transparency, the code of the vector addition example should be exactly the same as in the software-only case (except that now a part of the code responsible for the very computation is in synthesizable HDL). Figure 4 shows the code for the vector addition application in a paradigm that provides seamless hardware-software integration. It contains no explicit data partitioning and transfer, tasks delegated to the OS through a standard OS service (FPGA_EXECUTE). The programmer interface to hardware is clean and elegant.

To achieve this scenario, we implement a shared virtual-memory address space for both the software and the hardware threads (as in the software-only case in Figure 2b, except that now the slave thread is in hardware). The hardware thread (actually its HDL code) also generates virtual-memory addresses to access data. The system performs communication and synchronization using the same primitives as in the software-only case. The software is unaware that it communicates with a nonsoftware thread, and the hardware code, because it uses virtual addresses, is independent of the host platform. The programming is now transparent, and the hardware-software interfacing is portable. The memory access pattern required by the application (vector index access in this example) is no longer explicit in software, as it was in the wrapper code in Figure 3a.

```
/* Master Thread */
void main() {
        int *A, *B, *C;
        int thr_id;
        …
        read(A, SIZE);
        read(B, SIZE);
        thr_id = thr_create(add_vectors, A, B, C, SIZE);
        do_some_work_meanwhile();
        thr_join(thr_id);
        …
/* Accelerator Thread */
void add_vectors(int *A, int *B, int *C, int SIZE) {
        FPGA_EXECUTE(…);
}
```

Figure 4. Vector addition application code in seamless integration paradigm. The master thread is still unchanged, but there is no need for the wrapper thread; the application delegates accelerator control and data transfers to the OS through FPGA_EXECUTE. In application execution, the master (software) and accelerator (hardware) threads share the same virtual-memory address space, and the OS manages virtual-to-physical-memory-address translation for both software and hardware threads.

Therefore, accelerators are now implementable even when access patterns are unknown at compile time (as it is, for example, in pointer chasing, tree traversal, and random accesses): The OS now handles transfers at runtime.

Figure 5 summarizes the example to illustrate the benefits of our approach. In standard multithreading (Figure 5a), an abstraction layer, usually consisting of threading libraries (for example, Posix threads) and supporting OS services, provides for the simultaneous execution of software threads. But existing abstraction layers don't support integrating hardware accelerators with software (Figure

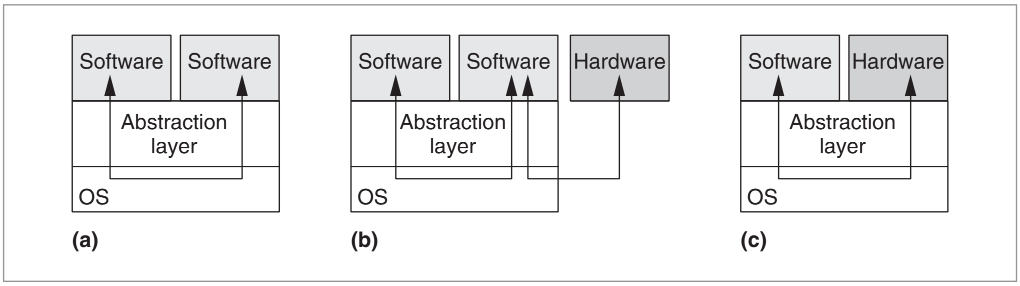


Figure 5. Standard multithreading (a) relies on an abstraction layer and usually some OS services. If the application includes a hardware accelerator (b), there is no systematic support. Instead, the programmer must use a wrapper thread. Extended multithreading (c) includes support for hardware accelerators in the abstraction layer and the OS, permitting seamless integration of software and hardware.

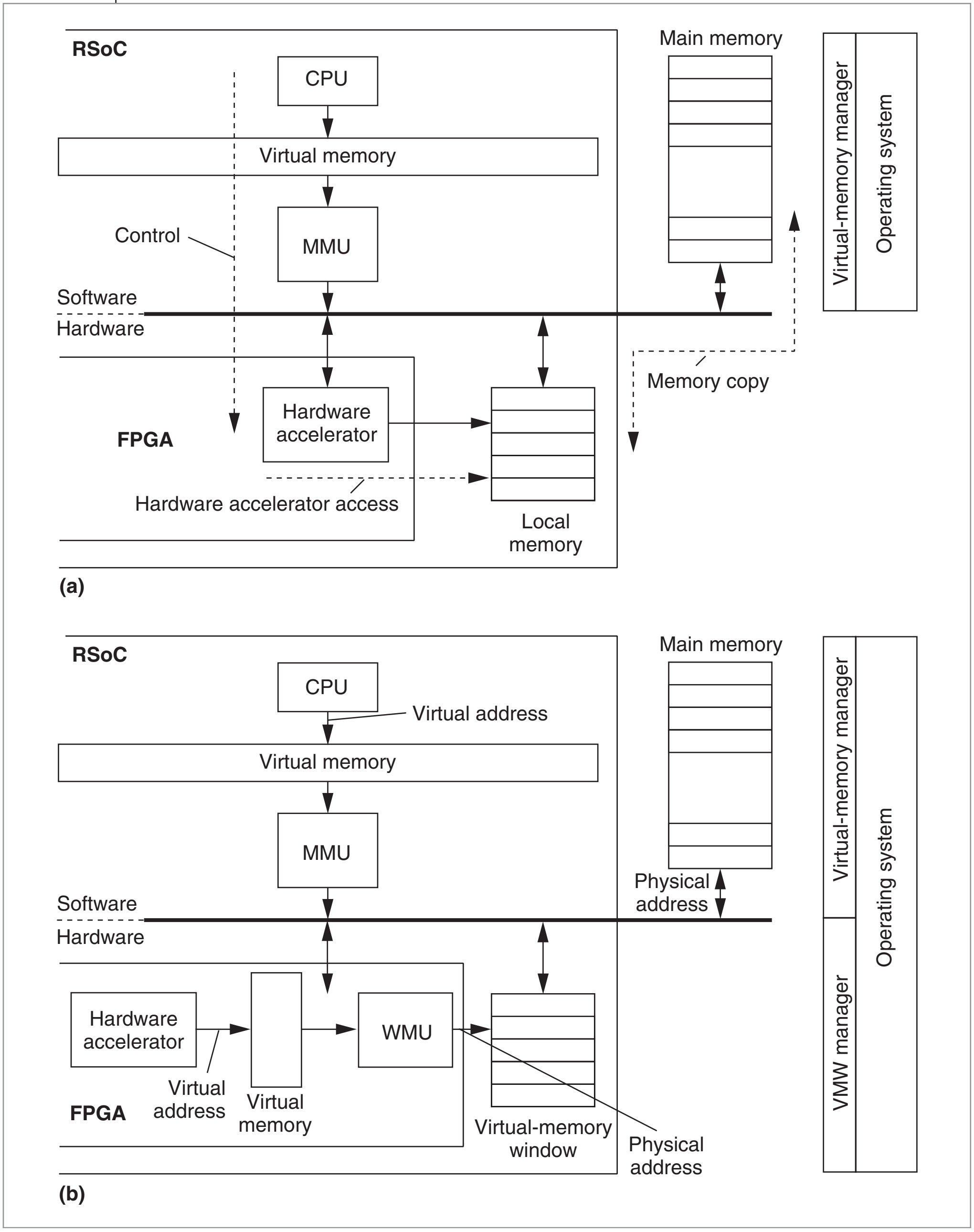


Figure 6. FPGA accelerators in an RSoC: typical (a) and with a virtualization layer (b) cases.

5b), and wrapper threads typically perform accelerator control and data transfers. Extending the abstraction layer (Figure 5c) to support hardware threads (that is, to enable communication, synchronization, and sharing of virtual-memory address space), simplifies programming and brings it to the software-only level.

Virtualization layer for transparent programming

We can achieve transparency and portability of reconfigurable applications by extending the abstraction layer—consisting of the thread library and OS support—to seamlessly support hardware threads. The extended abstraction layer—or virtualization layer—consists of both software and hardware components. To illustrate the concept in practice, we describe our implementation running on an RSoC platform.

Typical hardware accelerator

Figure 6a shows a typical RSoC, which can execute critical application parts in hardware. A standard CPU executes the user application, while the FPGA executes a hardware accelerator running on behalf of the user application. The software threads generate memory accesses to virtual memory, hiding all details of the physical-memory implementation. The memory management unit (MMU) performs translation from virtual- to physical-memory addresses.[6] The virtual-memory manager (VMM) supervises translation. The hardware accelerator generates the local memory's physical addresses and interfaces directly to the host platform. The programmer controls the accelerator by accessing its control/status registers, which are usually memory mapped. Then, the programmer schedules and performs the transfers from main memory to local memory and vice versa. Programming is thus neither transparent nor portable: Memory communication is explicit in the software code, and the accelerator's HDL code contains platform-specific details.

Virtualization extension

Figure 6b shows an RSoC with a virtualization layer.[7] As in the RSoC in Figure 6a, a standard CPU executes the user application, while the FPGA executes a hardware accelerator running on behalf of the user application. However, instead of directly interfacing to the host platform, the hardware accelerator communicates with the rest of the system by using the virtualization layer. The accelerator no longer generates physical-memory addresses. Instead, it generates virtual-memory addresses translated by a hardware translation engine called the window management unit (WMU), which is functionally similar to the MMU. The WMU translates virtual-memory accesses to physical addresses of local memory. Local memory is divided into pages forming a virtual-memory window (VMW).

Figure 7 shows how the OS provides the required data. If the accelerator generates an address of data not present in local memory, a VMW miss interrupt occurs. While the accelerator is stalled, the OS copies—invisibly for both user software and hardware—the required page from main memory to local memory. Once the OS finishes the transfer, the accelerator can proceed with the computation. Besides memory transfers, the VMW manager ensures memory consistency.

The virtualization layer consists of a software part (the VMW manager, which provides standardized OS services to the user space libraries and applications) and a hardware part (the WMU, which provides standardized hardware interfacing to the hardware accelerators). If a virtualization layer exists for a particular platform, the reconfigurable application becomes perfectly portable. To run it on a different reconfigurable platform, we need only recompile and resynthesize it.

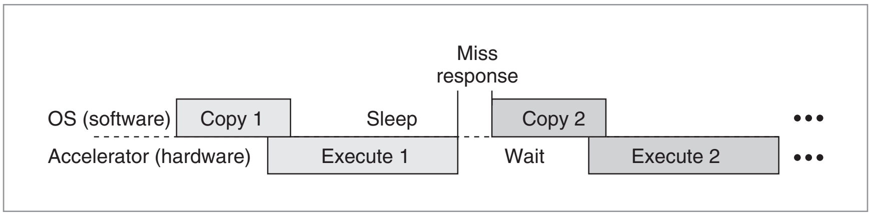


Figure 7. OS data transfers initiated by accelerator memory accesses. A hardware attempt to access data not present in local memory generates a miss. After a response time, the OS transfers the required page from main memory and resumes accelerator execution.

```
-- Initialization
-- with platform-dependent addresses
ptr_a <= BUF_ADDR;
ptr_b <= BUF_ADDR + BUF_SIZE/3;
ptr_c <= BUF_ADDR + 2*BUF_SIZE/3; …

-- Computation
cycle 1:
        -- partition of A[]
        BUF_ADDR <= ptr_a;
        BUF_ACCESS <= ’1’;
        BUF_WR <= ’0’;

cycle 2:
        reg_a <= BUF_DATAIN;
        -- partition of B[]
        BUF_ADDR <= ptr_b;
        BUF_ACCESS <= ’1’;
        BUF_WR <= ’0’;

cycle 3:
        reg_b := BUF_DATAIN;
        reg_c := reg_a + reg_b;
        -- partition of C[]
        BUF_ADDR <= ptr_c;
        BUF_DATAOUT <= reg_c;
        BUF_ACCESS <= ’1’;
        BUF_WR <= ’1’;
        ptr_{a,b,c} <= ptr_{a,b,c} + 1;
        if (ptr_c = BUF_ADDR+BUF_SIZE) then
        -- finished for a data chunk
            partial_finish;
        else
            cycle 1;
        end if;
```

(a)

```
-- Initialization
-- with runtime-dependent
   virtual-memory pointers
ptr_a <= A;
ptr_b <= B;
ptr_c <= C; …

-- Computation
cycle 1:
        -- object A[]
        VIRTMEM_ADDR <= ptr a;
        VIRTMEM_ACCESS <= ’1’;
        VIRTMEM_WR <= ’0’;

cycle 2:
        reg_a <= VIRTMEM_DATAIN;
        -- object B[]
        VIRTMEM_ADDR <= ptr_b;
        VIRTMEM_ACCESS <= ’1’;
        VIRTMEM_WR <= ’0’;

cycle 3:
        reg_b := VIRTMEM_DATAIN;
        reg_c := reg_a + reg_b;
        -- object C[]
        VIRTMEM_ADDR <= ptr_c;
        VIRTMEM_DATAOUT <= reg_c;
        VIRTMEM_ACCESS <= ’1’;
        VIRTMEM_WR <= ’1’;
        ptr_{a,b,c} <= ptr_{a,b,c} + 1;
        if (ptr_c = C+SIZE) then
        -- finished for the entire vectors
            finish;
        else
            cycle 1;
        end if;
```

(b)

Figure 8. VHDL-like hardware accelerator code: In platform-dependent code (a), the hardware accelerator generates physical addresses. In portable code (b), the hardware accelerator generates virtual-memory addresses, and the abstraction layer provides translation and synchronization.

Returning to our simple example, Figure 8 contrasts the HDL-like code of the vector addition application for the RSoC platforms in Figure 6. The code in Figure 8a generates platform-dependent addresses to access the buffer region. The buffer has a hard-coded size and address in the system memory map. Should the target platform change, along with its buffer size and memory mapping, the code would require modification. Instead of generating physical buffer addresses, the accelerator can use vir-

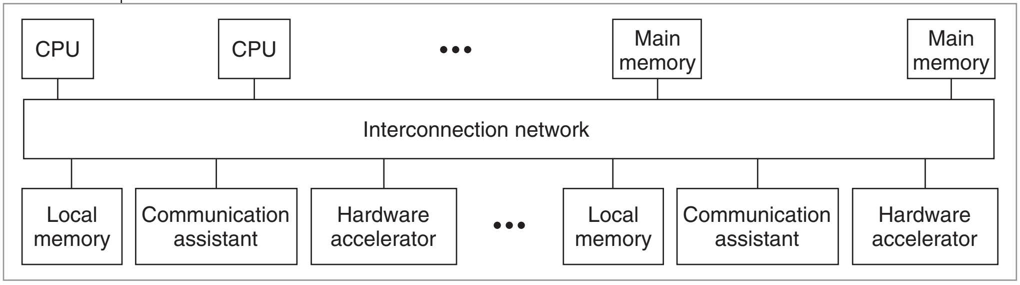


Figure 9. Reconfigurable parallel architecture: Computing nodes (standard processors and hardware accelerators) communicate through a general interconnection network. Hardware accelerators interface to local memories and the rest of the system through standardized communication assistants (CAs).

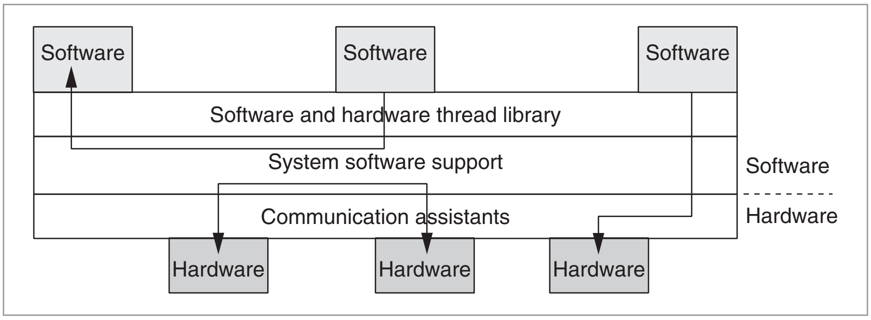


Figure 10. Software and hardware threads connect through the virtualization layer.

tual ones (as Figure 8b shows) that belong to the application address space. The accelerator code thus embodies only pure functionality and communicates through the standard shared-virtual-memory paradigm.

General virtualization architecture

Generalizing the architecture shown in Figure 6b leads to a reconfigurable parallel computer similar to a general parallel computer.[8] Besides general-purpose processors, this reconfigurable computing system contains application-specific accelerators implemented in—but not necessarily limited to—reconfigurable hardware and running on behalf of a particular application. Figure 9 presents such an architecture, in which general-purpose processors and reconfigurable hardware accelerators communicate through a general interconnection network.[9]

A local memory at each hardware accelerator node is accessible directly by the node and indirectly by others through the network. Each reconfigurable node contains a hardware interfacing component that is actually a standardized communication assistant (called the WMU in the practical example described earlier). The communication assistant defines the hardware interface (signals and protocols) and translates virtual to physical addresses. It guarantees correct execution either by initiating a copy to local memory or by accessing remote data.

Figure 10 shows the general virtualization layer, which supervises the copying and remote accesses. It provides transparent, platform-independent communication of software and hardware threads. Programmers and hardware designers can compile, synthesize, and run a multithreaded application on any reconfigurable platform provided that the platform has an implemented virtualization layer. The virtualization layer consists of software (libraries and system software) supported by hardware (communication assistants). Its task is to abstract details of the underlying architecture. Application threads are unaware of the data's physical location. Whatever address an application accesses, the virtualization layer manages transparent data movement and interthread communication. A virtualization layer can use various memory consistency protocols.[8] As in software-only multithreading, application architects should be aware of the potential penalties of coarse-grained data sharing. The virtualization layer can try to minimize these penalties through both hardware and software.

Results

We implemented a simplified virtualization layer (currently supporting only a single hardware accelerator) on a real RSoC system: an Altera Excalibur (EPXA1)-based board running the Gnu/Linux OS.[7] The virtualization layer consists of a Linux OS module (called the VMW manager) with some hardware support (the WMU from Figure 6b). The virtualization layer allows a hardware accelerator running on behalf of a user application to access the user space virtual memory. It also provides hardware-agnostic software functions; programs can invoke functions executed in hardware (hardware accelerators) as if they were common software functions.

To prove our approach's viability, we ported two

```
/* main function */
void main() {
        char *d_in; short *d_out; int size;
        …
        read(d_in, size);
        adpcm_decode(d_in, d_out, size*2);
…
}

/* library function */
int adpcm_decode(char *d_in, short *d_out, int_size) {
        struct cp_param param;
        …
        param.u.nparam = 3;
        param.p[0] = d_in;
        param.p[1] = d_out;
        param.p[1] = size;
        FPGA_EXECUTE(HW_ADPCM, &param);
        return param.u.retval;
}
```

(a)

```
/* main function */
void main() {
        int *A, *B, n64;
        …
        read(A, n64);
        idea_encrypt(A, B, n64);
        …
}

/* library function */
int idea_encrypt(int *A, int *B, int n64) {
        struct cp_param param;
        …
        param.u.nparam = 3;
        param.p[0] = A;
        param.p[1] = B;
        param.p[2] = n64;
        FPGA_EXECUTE(HW_IDEA, &param);
        return param.u.retval;
}
```

(b)

Figure 11. Programming with the virtual-memory window: ADPCM voice decoding (a) and IDEA cryptography (b) applications. The main function initializes the input data and calls the appropriate library function. The library function calls the hardware accelerator through an OS service and passes the virtual-memory pointers to hardware.

applications to the platform: the International Data Encryption Algorithm (IDEA) and adaptive differential pulse code modulation (ADPCM) voice decoding, a common multimedia benchmark. We implemented the critical parts of both applications in VHDL as coprocessors using the WMU interface. Figure 11 shows a programming example for the two applications, similar to the one shown in Figure 4. It represents how calls to hardware are realized. The main function calls the hardware to perform the accelerated function. The library functions (idea_encrypt and adpcm_decode) initialize parameter-passing structures and pass virtual-memory pointers to the hardware accelerators. A standardized OS service (FPGA_EXECUTE) invokes hardware execution. Once the hardware finishes, the VMW manager returns control to software. Although the application code here is not multithreaded, the important point is that the virtualization layer hides all communication details so programming is transparent and portable. The layer performs memory transfers and ensures consistency with no programmer interaction.

Figure 12 presents the two applications' execution times. The figure shows results for pure software, a typical coprocessor (a hardware accelerator directly programmed from the user application, similar to those shown in Figures 3 and 6a), and a VMW-based coprocessor. Small input data sizes allow both input and output data to fit into local memory accessed by the coprocessors.

Figure 12 shows that the coprocessors achieve significant speedups over software. The performance penalty of the overhead introduced by the virtualization layer is limited, as the differences between the two types of coprocessors' execution speedups show. The penalty is mostly due to the fact that we implement this WMU in an FPGA. In principle, however, we could implement the WMU in an ASIC as a standard SoC part, exactly as an MMU is today a standard hard-wired component in every chip. In that case, the overhead would become negligible.

Table 1 shows the WMU's complexity in terms of occupied FPGA resources (logic cells and memory blocks) for the EPXA1. Although the FPGA device we used is the smallest in its family, the WMU overhead is acceptable (the WMU uses no more than one-fifth the EPXA1's resources). The WMU fraction columns show the portion of the two accelerators' overall designs occupied by the WMU. For complex VMW accelerator designs such as IDEA, the WMU's resource overhead is less significant. Moreover, area overhead would be practically null if the WMU were implemented in ASICs as a standard part of the SoC, and execution time overhead would decrease considerably.

Further benefits of virtualization

Virtualization represents an additional layer on top of already existing layers such as memory hierarchies and communication protocol layers; thus, it inevitably introduces overhead. Like typical layering approaches

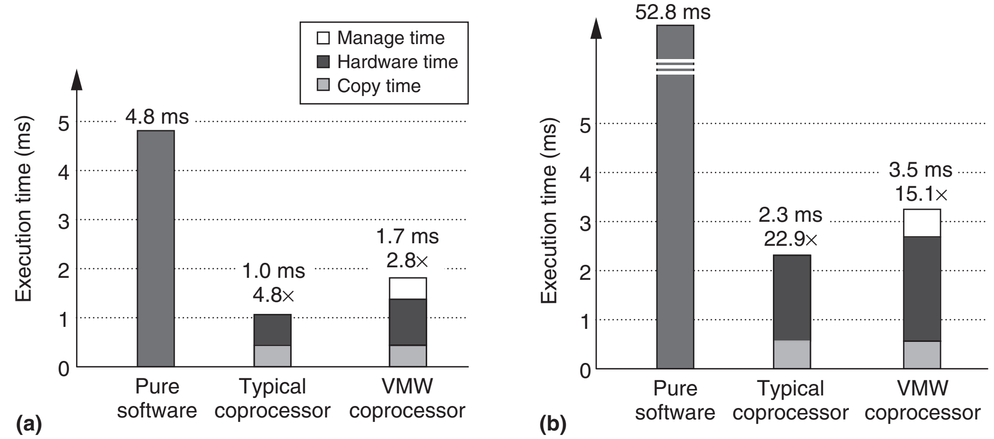


Figure 12. ADPCM (a) and IDEA (b) execution times. Coprocessor execution time consists of copy time (spent in software for transferring data to and from local memory), hardware time (spent in hardware for processing data), and manage time (spent in software for managing system data structures). Input data size is 2 Kbytes for ADPCM and 8 Kbytes for IDEA.

Table 1. WMU area overhead.

	FPGA (EPXA1) resources		Total accelerator resources	
Block type	**No. of units**	**WMU area (%)**	**WMU area, ADPCM (%)**	**WMU area, IDEA (%)**
Logic cell	576	14	48	16
Memory	5	19	83	45

such as virtual-memory abstraction, our approach basically trades efficiency for generality and programming comfort. Nevertheless, virtualization offers further benefits and does not always jeopardize efficiency.

Dynamic optimization

As it participates in application execution and communication transactions, the virtualization layer can dynamically decide when and where to improve execution. For example, with communication assistants extended by limited hardware, it can detect memory access patterns generated by hardware accelerators.[10] It can then predict future memory accesses and employ an adequate prefetching technique, attempting to hide memory communication latency.

Figure 13 shows how to improve application execution performance without programmer intervention or even knowledge. With no prefetching (Figure 13a), the hardware accelerator cannot execute continuously because of misses: When the required data is not present in local memory, the accelerator must wait for the OS to supply it; the OS fetches the page containing the required data and resumes the accelerator's execution. Between hardware executions, the OS part of the virtualization layer spends some time idle (sleep time). We can invest this time in a dynamic optimization (Figure 13b). With the communication assistant extended to detect a sequential memory access pattern, the virtualization layer can predict that future accesses will also be sequential and supply data partitions likely to be accessed in subsequent execution phases. With some additional hardware, the virtualization layer can verify its past predictions and decide whether to continue or bail out. Prefetching is not limited to sequential accesses, and several well-established prefetching techniques for nonsequential accesses are available.

In practical cases, for applications that exhibit regular memory accesses, significant performance improvements are possible. For example, Figure 14 shows execution times for the accelerator version based on an improved VMW (with dynamic prefetching implemented in the virtualization layer[10]). Although running at the same speed, the ADPCM coprocessor with prefetching finishes its task almost twice as fast as without prefetching. As Figure 13 indicated, sleep time decreases because the OS is busy predicting future memory accesses. Correct predictions can dramatically minimize the number of page misses. For example, in the ADPCM case, for the input data size of 64 Kbytes and the page size of 4 Kbytes, the number of misses decreases from 80 without prefetching to only two with prefetching. Figure 14b shows the total execution times of IDEA encryption for different numbers of local memory pages (a VMW design parameter). Prefetching significantly improves IDEA execution time. With prefetching, management time is slightly longer than without prefetching. With smaller page sizes, manage and copy time intervals become comparable to the hardware execution intervals: There are more pages in local memory, and thus there are larger data structures for the VMW to process. The number of pages chosen has some influence on performance.

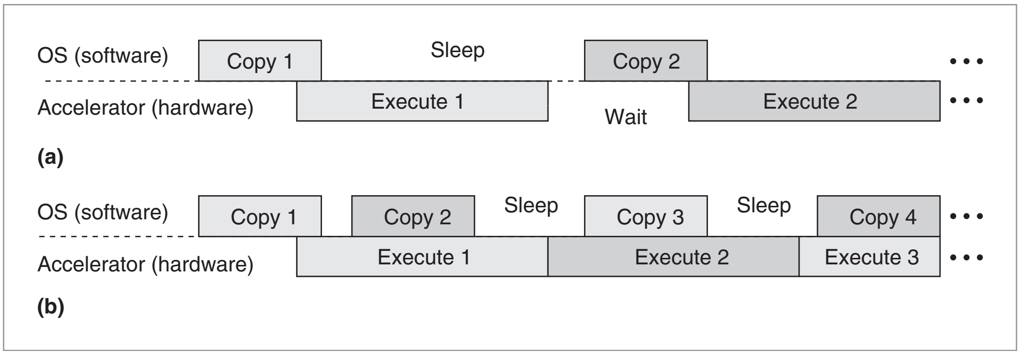


Figure 13. Execution timelines of software (OS) and hardware accelerator events without (a) and with (b) dynamic prefetching. In the prefetching case, the virtualization layer (in the OS) uses some idle time to try predicting future memory accesses and speculatively supplying data in advance. This can provide uninterrupted execution of hardware accelerators.

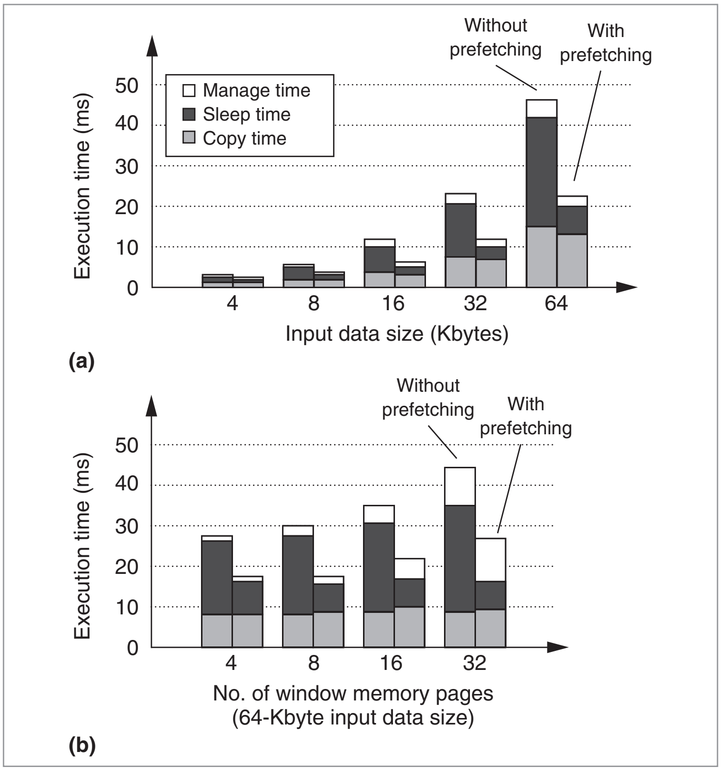
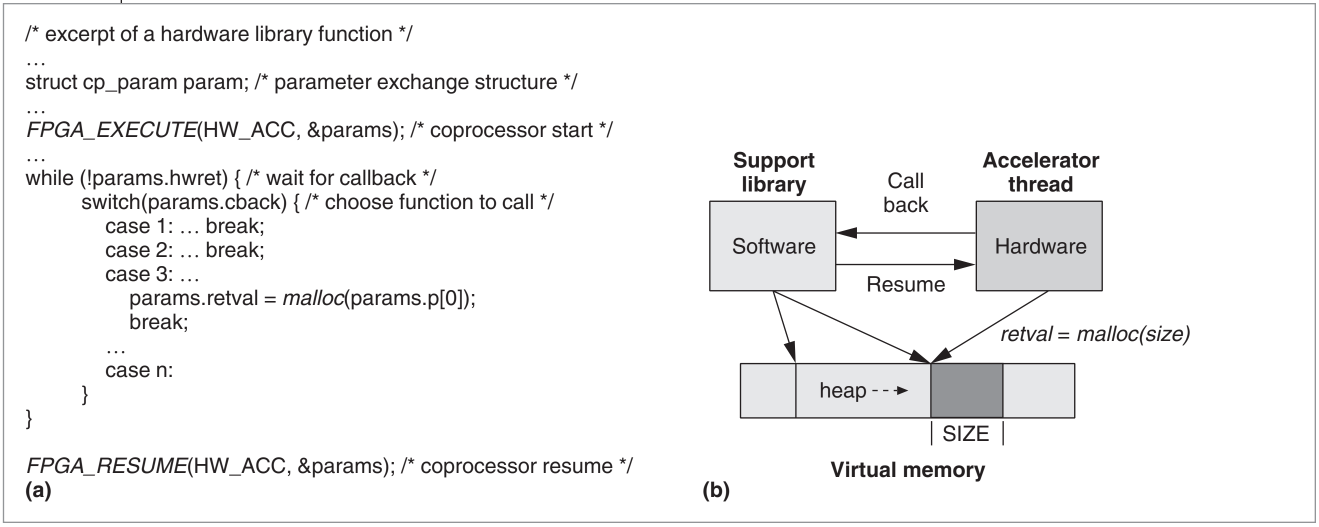


Figure 14. Virtualization layer with and without dynamic prefetching: total execution times of ADPCM for different input data sizes (a); total execution times of IDEA for different numbers of local memory pages (b). Execution time consists of copy time (for transferring data to and from local memory), sleep time (the OS is idle, and the accelerator is possibly processing data), and manage time (for managing VMW data structures).

With dynamic prefetching, VMW-based applications could even outperform typical coprocessor solutions. For example, in the IDEA application, for an input data size of 64 Kbytes, the typical coprocessor provides a 22× speedup, whereas the VMW-based coprocessor with dynamic prefetching would provide a 24× speedup. The VMW-based application can achieve such a speedup thanks to the implementation of dynamic optimizations within the virtualization layer, without any changes by the software programmer and the hardware designer. Execution of the application code (both software and hardware) used in the nonprefetching case improves simply because of the enhanced virtualization layer, requiring neither recompilation nor resynthesis.

Unrestricted automated synthesis

Hardware synthesis from high-level programming languages has long been a challenging topic. Translating simple control and pure dataflow segments of high-level programs into hardware is straightforward. In contrast, mapping advanced high-level language concepts and specific features to hardware is usually difficult. For example, accessing memory in C using pointers—possibly aliased pointers—complicates hardware synthesis.[11] Also, a programming concept like



```
/* excerpt of a hardware library function */
…
struct cp_param param; /* parameter exchange structure */
…
FPGA_EXECUTE(HW_ACC, &params); /* coprocessor start */
…
while (!params.hwret) { /* wait for callback */
        switch(params.cback) { /* choose function to call */
            case 1: … break;
            case 2: … break;
            case 3: …
                params.retval = malloc(params.p[0]);
                break;
            …
            case n:
        }
}

FPGA_RESUME(HW_ACC, &params); /* coprocessor resume */
```

Figure 15. Library code for malloc callback (a). In code execution (b), the accelerator calls back the software. The library code calls the appropriate function, and it passes the returned virtual-memory pointer to the accelerator. Hardware accelerator execution then resumes.

recursion can pose problems in synthesizing hardware. The virtualization layer's existence fosters and significantly simplifies automated synthesis.

Figure 8 indicated that there is no need for special treatment of pointers. The virtualization layer enables hardware threads to generate virtual memory addresses belonging to the application address space. Suppose that a pointer synthesized into the accelerator is eventually aliased, and that the accelerator uses the aliasing pointer to access the pointed-to data. The virtualization layer can transparently handle the access by copying the data to the access-generating node and, at the same time, take care of memory consistency (for example, write access by the aliasing pointer can initiate data invalidation at the accelerator's local memory). This is a common multiprocessor OS feature.

Figure 15 illustrates how the virtualization layer enables a hardware accelerator to call back software, even for sophisticated functions such as heap memory allocation (malloc from the standard C library). This software callback feature further facilitates synthesis and hardware-software partitioning. The library code performs the function call appropriate to the function identifier passed by the hardware. Malloc returns a virtual pointer to the newly allocated memory. Through the virtualization layer, the library passes it to the hardware accelerator. When the accelerator resumes, it can use the pointer without obstacles; the generated addresses will initiate transfer of the accessed data to the accelerator's local memory. No actions are required by the programmer or the hardware designer. However, an agreement (standardized by the VMW) must exist between software and hardware that the library will call the appropriate software function for the callback identification passed by the accelerator.

The virtualization layer can also appropriately treat recursion, as well as dynamic thread creation, thus significantly simplifying synthesis. Imagine a hardware thread that, either directly or indirectly, calls itself. Because every call goes through the virtualization layer, the latter can dynamically decide how to proceed. If the accelerator supports recursion (for example, if it has an internal stack to preserve the state), the virtualization layer will pass parameters and return control to the accelerator. Otherwise, the virtualization layer can call a software equivalent of the accelerator, thus dynamically changing the execution manner from spatial to temporal. Once recursion is finished, the shared memory mechanism automatically reflects changes back to the originating hardware thread.

The existence of a virtualization layer allowing unhindered hardware synthesis also facilitates easier software-to-hardware migration, even dynamically, during runtime. Partitioning applications to software and hardware is easier because synthesis is unlimited.[12] A virtualization-layer-enabled reconfigurable platform can be an ideal test bed for design space exploration and prototyping. With an automated synthesizer available, the designer can quickly switch from one possible solution to another. The virtualization layer respects the encapsulation principle (that is, software is unaware of hardware accelerators, which appear as if they were software), so even runtime changes of computation manner (temporal or spatial) are possible and invisible to the rest of the application.

In our current work, we are implementing a system with the full-fledged multithreaded programming paradigm described here. We plan to explore other dynamic optimizations in the virtualization layer, especially prefetching techniques for nonsequential memory accesses. We are also addressing unrestricted automated synthesis: We have built a basic synthesis flow for our platform and are considering how to extend the synthesizer to cover advanced high-level language concepts. ■

■ References

1. S. Vassiliadis et al., “The MOLEN Polymorphic Processor,” *IEEE Trans. Computers*, vol. 53, no. 11, Nov. 2004, pp. 1363-1375.
2. G. Brebner, P. James-Roxby, and C. Kulkarni, “Hyper-Programmable Architecture for Adaptable Networked Systems,” *Proc. 15th Int’l Conf. Application-Specific Systems, Architectures and Processors* (ASAP 04), IEEE Press, 2004, pp. 328-338.
3. D. Andrews et al., “Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link,” *IEEE Micro*, vol. 24, no. 4, July-Aug. 2004, pp. 42-53.
4. B. Nichols, *Pthreads Programming: A POSIX Standard for Better Multiprocessing*, O’Reilly and Associates, 1996.
5. A.S. Tanenbaum, *Modern Operating Systems*, 2nd ed., Prentice-Hall, 2001.
6. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2002.
7. M. Vuletić, L. Pozzi, and P. Ienne, “Virtual Memory Window for Application-Specific Reconfigurable Coprocessors,” *Proc. 41st Ann. Conf. Design Automation* (DAC 04), ACM Press, 2004, pp. 948-953.
8. D.E. Culler, J.P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, 1999.
9. M. Vuletić, L. Pozzi, and P. Ienne, “Programming Transparency and Portable Hardware Interfacing: Towards General-Purpose Reconfigurable Computing,” *Proc. 15th Int’l Conf. Application-Specific Systems, Architectures and Processors* (ASAP 04), IEEE Press, 2004, pp. 339-351.
10. M. Vuletić, L. Pozzi, and P. Ienne, “Dynamic Prefetching in the Virtual Memory Window of Portable Reconfigurable Coprocessors,” *Proc. 14th Int’l Conf. Field-Programmable Logic and Applications*, LNCS 3203, Springer-Verlag, 2004, pp. 596-605.
11. L. Semeria, K. Sato, and G. De Micheli, “Synthesis of Hardware Models in C with Pointers and Complex Data Structures,” *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 6, Dec. 2001, pp. 743-756.
12. C. Dubach, *Java Byte Code Synthesis for Reconfigurable Computing Platforms*, master’s thesis, Ecole Polytechnique Fédérale de Lausanne, Switzerland, 2005.

Miljan Vuletić is pursuing a PhD in the School of Computer and Communication Sciences at the Ecole Polytechnique Fédérale de Lausanne (EPFL). His research interests include reconfigurable computing and operating systems. Vuletić has a BSc in electrical engineering from the University of Belgrade, Serbia and Montenegro. He is a student member of the IEEE and the ACM.

Laura Pozzi is a postdoctoral researcher in the School of Computer and Communication Sciences at the Ecole Polytechnique Fédérale de Lausanne (EPFL). Her research interests include automating embedded-processor customization, high-performance compiler techniques, and reconfigurable computing. Pozzi has an MS and a PhD, both in computer engineering, from the Politecnico di Milano, Italy.

Paolo Ienne is a professor in the School of Computer and Communication Sciences at EPFL, where he heads the Processor Architecture Laboratory. His research interests include aspects of advanced SoC design such as automatic processor specialization, programming abstractions for reconfigurable computing, and self-calibrating design methodologies. Ienne has an MS in electrical engineering from the Politecnico di Milano and a PhD in computer science from EPFL. He is a member of the IEEE and the IEEE Computer Society.

■ Direct questions and comments about this article to Miljan Vuletić, EPFL-IC-ISIM-LAP, Station 14, 1015 Lausanne, Switzerland; miljan.vuletic@epfl.ch.