CHAPTER 11

OPERATING SYSTEM SUPPORT FOR RECONFIGURABLE COMPUTING

Katherine Compton
Department of Electrical and Computer Engineering
University of Wisconsin–Madison

André DeHon
Department of Electrical and Systems Engineering
University of Pennsylvania

As part of the evolution of the field of reconfigurable computing, researchers are increasingly focusing their attention on the issues of integrating reconfigurable computing into multipurpose or general-purpose compute environments. Operating systems (OSs) fill two key roles in computing: simplifying the programming interface through an abstracted programming model and managing shared resources [40]. Both are critical to reconfigurable computing systems, which have in the past suffered from the stigma of programming difficulty as well as from a general focus on single-application systems and nonscalable, nonportable designs.

An operating system, coupled with the proper compilation environment, can simplify the programming of reconfigurable computing systems by providing a well-defined, well-documented compute model that abstracts the structure and capacity of the underlying hardware. This model may explicitly provide constructs for defining hardware tasks (the parts of the application implemented in reconfigurable logic). Alternately, it may be agnostic to the implementation medium. Like the compute fabric, the communication structures between tasks can be abstracted by the compute model to simplify the design process.

Reconfigurable hardware in a reconfigurable computing system is explicitly intended to be a shared resource. Even in a single-application system, hardware may be shared within the application to accelerate different tasks at different times. In a multitasking system, different threads of computation may vie for the hardware resources. The operating system arbitrates hardware use both within and across applications. Furthermore, the OS also provides protection and security to prevent a maliciously or poorly programmed application from compromising the system. Through isolation, the operating system also provides a safe environment where applications can be debugged and inspected without concern that buggy code will affect system stability.

The demands on an operating system for reconfigurable computing include

- Abstraction of the capacity and composition of reconfigurable hardware resources.
- Scheduling use of shared resources across processes.
- Methods for communication and synchronization among hardware tasks and software.
- Protection of the tasks of one process (hardware and software) from those of another.

This chapter discusses the above concepts in terms of both key roles of an operating system: the programmer's view and the management of shared resources.

11.1 HISTORY

Although the concept of operating system support for reconfigurable computing has existed since at least 1996 [6], the idea languished for a time, not quite gaining popular momentum. A significant barrier to operating system development for reconfigurable computing has been the lack of a standard reconfigurable computing hardware platform as a focus for commercial and academic development.

With much of reconfigurable computing research focused on specialized scientific computers or embedded systems, researchers were willing to forgo the abstraction/virtualization benefits provided by an operating system. Instead, application designers (who frequently were the hardware/system designers) would include hardware management operations in their application, explicitly deciding when and where to load particular operations. Manual management leveraged the designer's understanding of the application to provide potentially better performance than an OS layer, discouraging many researchers from dedicating valuable research time to finding a more generic (but possibly less optimized) solution. Yet these systems too would benefit from operating system support to attract a broader group of application designers uninterested in every hardware detail or in micromanaging its use. Even those with suitable hardware backgrounds could then focus their efforts on application (instead of hardware) details.

The increase in demand for operating system support is mirrored, in part, by the increase in complexity of embedded systems and applications. Many single-function devices of the past have evolved into multifunction devices. Cell phones, for example, not only provide basic voice communication, but also capture pictures and video, replay video and audio, browse the Internet, communicate with other electronic devices, and support gaming. A device may execute several of these applications over time, giving it a "general-purpose" flavor within an "embedded" body. Reconfigurable hardware is attractive for devices such as this because of its flexibility to reconfigure to accelerate a variety of applications. The compute-intensive computations of an application execute

in hardware to operate faster, using less power (battery) than even an embedded instruction-based processor [34,41].

Even a single-function device may require many different compute-intensive operations. For example, a digital audio player may need to perform error checking, Huffman decoding, IDCT, and other tasks. The reconfigurable hardware that accelerates these operations may, because of cost considerations, be too small to fit all hardware tasks simultaneously. However, an operating system can automatically reconfigure it to implement each task in sequence, as shown in Figure 11.1 (and discussed in Chapters 4, 5, and 9), allowing applications to execute all hardware tasks as if they were persistent in hardware. This provides the application programmer with a virtualized hardware view not hampered by low-level details.

Another contributor to the growing demand for OS support for reconfigurable computing is the increasing difficulty of providing clock speed increases to general-purpose processors [1]. This problem is causing researchers to more closely investigate the potential benefit of reconfigurable computing in general-purpose computers in order to boost performance for compute-intensive applications, including multimedia and communications applications. Using reconfigurable computing in a general-purpose machine requires more

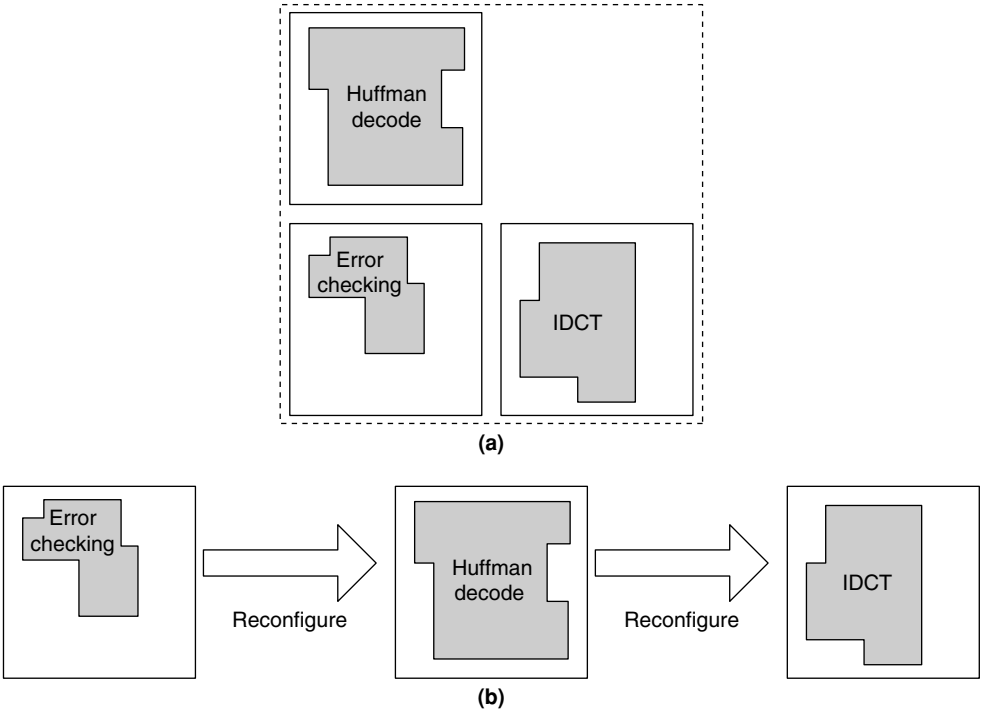

FIGURE 11.1 ■ The abstraction of a large virtual hardware capacity (a) can be implemented on more limited hardware resources using runtime reconfiguration (b).

sophisticated resource management than can be expected from individual applications, further driving the need for OS support.

11.2 ABSTRACTED HARDWARE RESOURCES

The official Commodore *Programmer's Reference Guide* for the C64 computer, originally published in 1982, provides programmers with a great deal of information [10]. For example, it contains a table of the memory map of the C64, including the memory location of the BASIC ROM, the memory-mapped screen output, other memory-mapped I/O, and available program memory locations. It even provides the pinouts of the C64's I/O ports and a schematic of the motherboard—as information to a *programmer*. Much like the preceding evolution of abstracted programming models for mainframe computers [28], increased complexity in personal computing systems later both enabled and required an increase in abstraction.

Today's programming texts do not provide explicit hardware details; instead, for example, they instruct on the use of system calls to provide I/O. One can write an application in a high-level language such as Java or C without knowing even what processor it will run on or how much memory the system will have. For some time, the average software programmer has not needed an understanding of underlying hardware.[1]

To ease programmer burden, the OS provides an abstracted view of hardware—a simpler *virtual* machine as the target for the application. In this virtual machine, the programmer may use library or system calls that provide standardized interfaces to interact with a wide variety of I/O, such as the screen, storage units, and other peripherals. The virtual machine also gives the programmer the appearance of isolation, effectively providing the illusion of dedicated use of the computer's resources [47]. Furthermore, specific details of system resources, such as their quantity and speed, are abstracted. In reality, the operating system is managing these limited physical resources both to allow sharing and to avoid conflicts between applications, each of which was designed as if it were the only one in the system and as if resources were unbounded.

The section that follows discusses the abstraction provided to the programmer by the reconfigurable computing operating system.

11.2.1 Programming Model

Reconfigurable computing provides a mechanism for parallel computation. In some cases, compute-intensive tasks in a sequential application are converted to hardware to capture instruction- or data-level parallelism within the sequential framework. In others, the application is designed explicitly for parallel execution

[1] Embedded systems, some graphics, and other specialized programmers may still require some knowledge of hardware to target specific customized architectures or to meet stringent performance requirements.

throughout, with many concurrent hardware (and software) tasks. Chapter 5, Section 5.1, discusses a variety of compute models for reconfigurable computing.

The application developer, the compiler, or the runtime environment can create multiple interchangeable implementations for a task using a separable interface and implementation. This separation is analogous to the delineation between interface and architecture in VHDL (Chapter 6), or the interface and implementation in Java or C++. The operating system can use the interchangeable implementations to bind the computation to a specific resource at runtime, as discussed in Section 11.3. Compiled applications may be a combination of software components and either abstracted hardware components (which undergo the final steps of compilation/synthesis at install time or runtime) or configuration bitstreams that represent hardware tasks.

Depending on the development environment, designers may explicitly partition their application between hardware and software components, or the compiler may automatically partition a high-level application description (Chapter 26). If explicitly partitioned, the hardware components may be specified in a hardware description language (HDL) or in a high-level language with added constructs to specify parallelism, communication, variable bit width, or other hardware-specific features (e.g., Chapter 7).

Implicit partitioning facilitates application portability, as added language constructs for explicit partitioning may not be available for different systems, and hardware descriptions, while more portable than postsynthesis designs, may still depend on specific hardware features. Automatic partitioning and synthesis at compilation time allows an application description to be easily recompiled for different systems (provided that tool support is available).

Because software programmers are not usually hardware designers, and automatic compilation from a high-level language to hardware does not always provide acceptable results, reusable libraries can provide a balance between ease of specification and result quality. Developers can use library calls to perform compute-intensive operations without concerning themselves with how the operation is actually implemented (hardware versus software, hardware and software details). Libraries can contain efficient hardware implementations, potentially at multiple area/performance tradeoff points, and, possibly, software alternatives for a set of related operations [29, 45]. Static linking to such a library could significantly increase application distribution size if multiple implementation options were included to support different execution platforms or to provide runtime binding (as discussed in Section 11.3). A dynamically linked library (DLL) could ameliorate this problem if it were reused by other applications.

A final approach is to use description languages designed to be agnostic to the eventual implementation in hardware, software, or a mix of the two [13, 22]. Much of the automatic partitioning work focuses on high-level languages normally used in software programming, which were created for inherently sequential compute structures (instruction-based processors). Depending on the hardware design, a reconfigurable computing platform has the potential to provide much more parallelism at a variety of levels difficult to describe using

a software-centric approach. (These concepts are discussed in more detail in Chapter 5.)

Within an application, the programmer or compiler instantiates a hardware task as a virtual resource and later applies it to the suitable input data. When the operating system scheduler (Section 11.4) decides to allocate hardware to the task, it loads that task onto hardware. For best performance, a single hardware task can (and should) execute repeatedly on successive input data. Depending on the extent of runtime support, the operating system could instantiate multiple copies of the task to increase captured parallelism or time-multiplex multiple tasks if hardware resources are limited, as discussed in Section 11.3. This detail should be abstracted from the user, however, as the amount of resources available for the task can be based on runtime system state, which is likely unknown at design time. One approach (discussed in Chapter 9) is to design the application for *maximum possible* parallelism, with the operating system automatically time-multiplexing the different tasks if insufficient resources are available for the full application simultaneously [13].

11.3 FLEXIBLE BINDING

Because reconfigurable computing systems are inherently flexible, they allow the operating system greater freedom in managing shared resources. The operating system can perform flexible binding of tasks to different types of resources (hardware/software) and, for those bound to hardware, can perform a runtime tradeoff between resource use and performance. Flexible binding allows a single application to be implemented using different resources on different computing platforms, or even on the same platform at different times. Install-time binding decisions are based on the physical characteristics of the system (e.g., the number of programmable resources or memories). Runtime binding, on the other hand, uses information about the physical characteristics along with the current system state (e.g., number of running tasks) to make implementation decisions.

11.3.1 Install Time Binding

Install time binding involves the compilation of applications to a generic representation analogous to an intermediate representation in software compilation. Final synthesis of the generic representation occurs at install time based on the specific resource types available on the system. Install time binding is therefore important to the prevailing economic model of computer purchasing: Spending more money does not (generally) allow one to run *different* applications but rather the *same* applications *better*. Likewise, reconfigurable computing machines should be available at multiple price points, with the capacity/performance/power efficiency of their resources increasing in relation to cost.

Applications running on a more expensive, more powerful machine should perform better than those running on a base machine—but they should still *run* on that base machine. In keeping with this economic discussion, if the reconfigurable hardware in a computer is upgraded, the applications may require reinstallation to leverage the new resources. Depending on the level of abstraction of the specification, this may require CAD processing, which should be performed quickly (and potentially in the background when the system is idle) to avoid system slowdown, as discussed in Section 11.3.3 and Chapter 20. An alternate form of install time binding is dynamic linking to precompiled libraries of hardware (and software) task implementations [29]. Libraries can be compiled for different platforms and distributed with the OS as part of the hardware drivers.

11.3.2 Runtime Binding

Runtime binding is based on both physical characteristics and current system state, and may be performed as part of the scheduling process (Section 11.4). It modifies a task's implementation based on the resources allocated to it during scheduling. The most simple form of runtime binding supports relocation of hardware tasks to different regions of the hardware resources. Relocation (discussed in more detail in Chapter 4) facilitates concurrent residency and/or operation of multiple hardware tasks. It also affects task communication, discussed in Section 11.5.

Another form of runtime binding allows a given task to execute in either hardware or software depending on scheduling decisions [14, 29, 31], discussed in Section 11.4.3. Systems that permit dynamic binding can avoid stalling for hardware availability by proceeding with a software alternative for the task. Dynamic hardware/software binding at runtime requires either a task executable capable of running on hardware or software (e.g., [22]) or a pair of interchangeable hardware and software implementations [13, 14, 29]. To facilitate application design and debug, the two components should have identical functional behavior.

Runtime binding can allow hardware tasks to expand or contract to make use of the resources allocated to them by the scheduler, as discussed in Section 11.4. This ability allows tasks to be implemented on a variety of architectures, from low capacity to high capacity, to promote portability. Hardware tasks can also be modified based on system load, occupying fewer resources in a system under heavy load and more in a system under light load, as shown in Figure 11.2. In (a), task A is using fewer resources because of increased demand by other tasks. In (b), task A rebounds to more resources after task B is no longer needed. Task A's data rate is improved in (b) by the increased parallelism.

A task can occupy fewer resources by time-multiplexing its functionality, or more resources by unrolling or replicating [13]. Time-multiplexing a task requires storage to hold intermediate results between the temporal partitions. Performing time-multiplexing or expansion at runtime can be quite expensive, potentially involving a modified CAD flow, as discussed next in Section 11.3.3. Alternately, implementations at multiple area–performance (or power) tradeoffs can be created at compilation time, eliminating transformation overhead at runtime [14, 29].

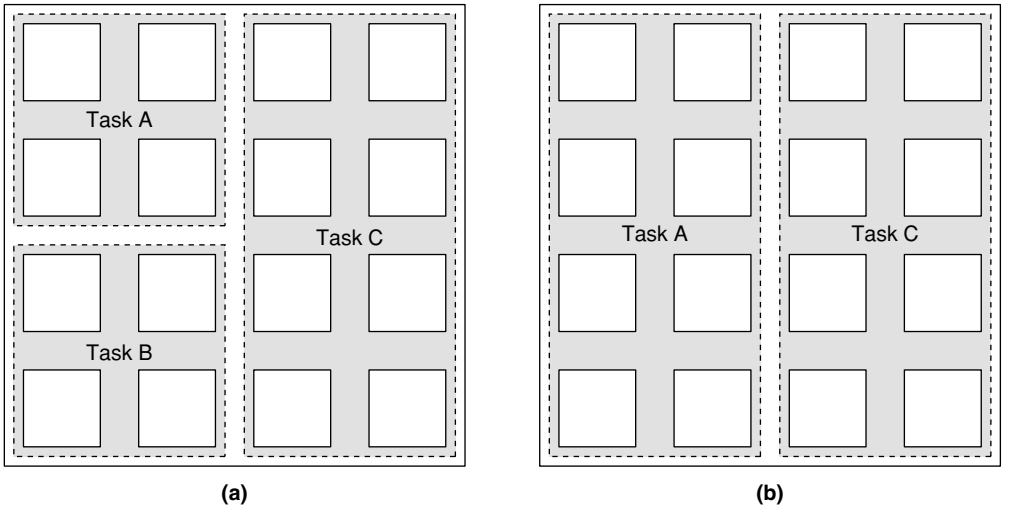

FIGURE 11.2 ■ Flexible binding allows tasks to use a different number of resources based on hardware capacity and resource availability. In this example, Task A can either occupy less area at the expense of performance (a), or achieve a higher data rate at the expense of area (b).

Although a specific palette of implementations reduces OS complexity, it also limits the possibilities of customizing the hardware task to the exact hardware resources available.

11.3.3 Fast CAD for Flexible Binding

Modifying a hardware task after application distribution may require that one or more CAD operations, such as placement, be applied at install time or runtime [13, 39, 43, 44] (e.g., Section 9.4). Unfortunately, CAD algorithms, depending on the problem size, can be quite slow. Chapter 20 discusses a number of fast CAD approaches for hardware task implementation motivated in part by flexible binding. Some possible solutions to accelerating install time or runtime CAD processes include

- Trading solution quality for speed in the CAD process (less optimized solutions).
- Accelerating CAD algorithms in hardware (i.e., implementing CAD hardware tasks on the target reconfigurable computing system).
- Abstracting some of the hardware detail to simplify the problem (applying algorithms to larger blocks of structures, where intragroup CAD decisions are fixed at compile time, and only intergroup CAD decisions are required at install time or runtime, as discussed in Chapter 4 and Section 9.2.4).
- Using a compile time CAD process to generate static information about the hardware task that can be used to accelerate later CAD operations (marking areas of the circuit for replication or time-multiplexing).

11.4 SCHEDULING

Scheduling determines what tasks should use hardware when, and may also decide how many resources (and what type) to allocate to each. These decisions may be made at compile time based on static application information, at runtime based on dynamic system status, or at a combination of the two. The scheduling goals may include maximizing application or system performance, minimizing power consumption, or meeting real-time deadlines. Achieving these goals also requires minimizing the reconfiguration overhead, as discussed in Chapter 4.

Schedulers that include resource allocation also perform flexible binding (Section 11.3), choosing specific resources to implement a given task and potentially altering that task to fit the resources. Flexible binding complicates the scheduler's decision process by expanding the search space. However, expanding the search space with flexible binding also opens the door to scheduling solutions that would otherwise not be possible.

11.4.1 On-demand Scheduling

One of the simplest forms of runtime scheduling is servicing hardware resource requests in the order received, reconfiguring as needed, and queuing requests that cannot yet be serviced [6]. When an application calls a hardware task, its request is sent to the operating system. If the task is preconfigured on hardware, it executes; otherwise, it must be loaded into hardware (configured) prior to execution. If all hardware resources are allocated and in use, the system will queue waiting requests until the resources are freed.

Hardware requests are generally blocking, forcing the requestor to busy-wait until the hardware is available. Then the task is configured and finally executes. Busy-waiting can contribute significantly to reconfigurable computing overhead, as discussed in Chapter 4, but the system (with an appropriate compute model) can use a sleep/wake approach instead of busy-waiting to allow nonblocking threads or processes to use the compute resources in the meantime, hiding some of the configuration latency. Furthermore, runtime binding, discussed in Section 11.3, allows threads or processes that might otherwise be blocked waiting for hardware availability to execute in software instead.

11.4.2 Static Scheduling

Static scheduling relies on analyzed, profiled, or annotated application behavior to determine when an application should request that each hardware task be configured [23, 26, 27]. Static schedulers operate "offline" and thus have a more global view of the task requirements and are able to search a greater expanse of the solution space than a dynamic (online) scheduler. Brute-force or Monte Carlo approaches may therefore be feasible for static schedulers even if prohibitively slow for dynamic scheduling. A static scheduler can also attempt to load hardware tasks prior to their execution to minimize configuration overhead (a technique known as prefetching [24]).

For static scheduling to be profitable, however, both the application task set and resource availability must be highly predictable. An offline schedule does not have access to runtime information and therefore cannot adapt to the current system load. This can prevent the static schedule from computing a good coschedule of multiple independent tasks. Further, if the static schedule is wrong about which tasks must run next, prefetching can actually be detrimental to performance, forcing needed configurations to be evicted and performing extra, unnecessary reconfigurations.

11.4.3 Dynamic Scheduling

Dynamic schedulers use runtime information to aid scheduling. Data-dependent application behavior, system load, and the characteristics of other executing applications can therefore all contribute to (and complicate) schedule computation. Although single-application behavior may be statically predictable in some cases, the interferences arising from multiple simultaneously executing applications lead to an explicitly nondeterministic interleaving of hardware task calls from different applications.

As a simplification, some schedulers use a window-based approach, dividing time into windows and solving the scheduling problem for each [14, 27, 31]. Figure 11.3 illustrates the timing of window-based scheduling. Once the scheduler determines which tasks should be implemented in hardware, the hardware must be reconfigured to implement them. After reconfiguration, the hardware can execute until the next reconfiguration phase in the following window. To minimize the impact of scheduling overhead, the window should be "large" compared to the time required to compute the schedule and perform reconfiguration. However, it should also be small enough to capture current system behavior for use in the scheduling decision. Statistics from the previous interval (or multiple previous intervals) provide recent behavior information to the scheduler.

A "frontier" dynamic scheduler [27] uses application dataflow and task execution information from the previous interval (such as which tasks executed and at what data rate) to compute the new schedule for the next interval. Input data availability and allocatable output space information are requirements for task scheduling and are used to compute the relative priority of tasks. The scheduler can use resource availability and information on data rate (of the considered task

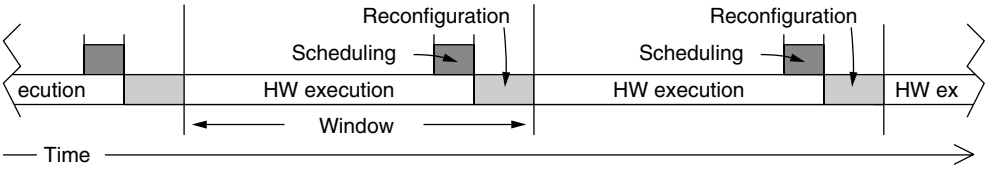

FIGURE 11.3 ■ A dynamic scheduler can divide time into a series of windows, each with its own scheduling problem.

and ones it communicates with) to choose a flexible binding implementation. This approach matches a streaming communication approach (Chapter 9), where good scheduling is necessary to minimize the buffering requirements between tasks. Flexible binding allows the frontier scheduler to time-multiplex or replicate tasks to balance the data rate between one task and those adjacent [27].

Runtime information, such as the frequency of task use in the prior interval and task performance information, also can be used without considering the dataflow of the executing applications. In each window, hardware resources can be treated as a knapsack, which the scheduler tries to pack with the greatest overall value [14]. Each task is assigned a "value" based on performance or power consumption and a "cost" based on hardware area requirements. Tasks not scheduled to hardware execute with lower performance in software to avoid starving less valued tasks. By including multiple implementations of a task with different values/costs, the scheduler can also use dynamic binding to adapt task implementations based on resource availability/demand [14, 27]. The knapsack problem can be solved either heuristically or, if the problem size is small enough, exactly.

11.4.4 Quasi-static Scheduling

A purely dynamic scheduler only considers information available at runtime and loses the opportunity to optimize based on known application characteristics. In contrast, quasi-static scheduling combines dynamic system and application information with static application analysis. Using dynamic management with static analysis enables the scheduler to more accurately predict near-future hardware task needs (and, just as important, which tasks will *not* be needed) [24, 27]. Quasi-static scheduling also accelerates the scheduling process by reducing the dynamic scheduler's burden.

For example, static analysis can provide the ordering of tasks within an application, timing estimates for when the tasks will be executed relative to one another, data rate analysis of different possible time-multiplexing/replications of the tasks, and intertask communication resource requirements. The runtime scheduler can then use dynamic scheduling techniques, but prune the solution space based on static analysis information to arrive at an improved solution more quickly. Dynamic scheduling can also allow otherwise statically scheduled applications to reuse hardware tasks configured for other applications to reduce configuration costs [35].

11.4.5 Real-time Scheduling

Scheduling for real-time systems considers task deadlines rather than general performance. Hard deadlines must be met within the specified time or the system has failed. An example of a hard deadline would be triggering operation of strictly timed automotive engine components. Soft deadlines must be generally met for acceptable use, but missing one or even a few is not mission-critical. An example of missing a soft deadline would be dropping a frame in real-time video.

Missing a soft deadline may not invalidate the computation, but it may degrade the application in some way. This type of operation is common in embedded systems. Indeed, real-time systems and their operating systems are the focus of much research [20, 23].

One approach to implementing reconfigurable real-time systems is to leverage the vast real-time research effort by wrapping hardware tasks with a thread interface [4]. Such a system includes a generic hardware-based scheduler for both hardware and software threads using whatever scheduling algorithm is implemented within it. Synchronization details of this approach are presented in Section 11.6.1.

Alternately, the scheduling algorithm can be tailored specifically to reconfigurable computing, using information about hardware capacity, task hardware requirements, and task configuration time in addition to deadline information [39, 43]. For example, tasks that can fit in a currently available area are more likely to be guaranteed to meet a deadline than are those that require reconfiguration due to reconfiguration overhead. If sufficient resources are free, but are distributed throughout the hardware, defragmentation may be required (see Chapter 4) to consolidate sufficient free space for the incoming task.

The time required for this process affects the ability of the system to meet the task's deadline. If free space is not available even with defragmentation, the task may be rejected or its deadline not guaranteed. The task could meet the deadline if one or more other tasks executing on hardware complete with enough time left to permit configuration and execution of the new task before its deadline expires. Alternately, a task implemented in hardware may be preempted (see next section) in favor of an incoming task if the latter has higher priority [43].

11.4.6 Preemption

A scheduler may use preemption to reallocate hardware to a "more desirable" task, whether based on meeting specific deadlines in a real-time system, based on the relative priority of different tasks in a performance-based system, or to allow a more balanced use of hardware in the presence of long-executing tasks [2, 18, 31, 43]. The configuration data for a given task holds some of the required information, such as circuit structure, and possibly initial values for embedded memories. However, any values in state-holding elements that change in response to hardware operation are not included. Therefore, the complete "saved state" for preempting a hardware task is a combination of its configuration data and the current values of any state-holding elements modified during execution. Provided a hardware interface to this information is available, the operating system can read the current state to store in memory and later load it back into hardware when needed.

Preemption is complicated by flexible binding if the implementation saved does not match the implementation resumed. The sizes of configuration data and the number of state-holding elements may not match between different implementations. Therefore, systems supporting flexible binding and preemption must save an abstracted view of task state.

11.5 COMMUNICATION

A key feature of communication abstractions is that they are, in fact, abstractions. Although certain abstractions may map well to specific hardware architectures (and vice versa), the use of one in particular does not necessarily require a particular hardware structure (or vice versa). For example, a message-passing abstraction could be implemented on a shared memory architecture, or a shared memory abstraction could be implemented on top of a message-passing architecture. Library calls and the compilation environment map communication abstractions to the actual implementation, and the operating system manages the implementation. The abstractions, however, allow the programmer to ignore implementation details and focus on efficient specification. The following subsections discuss a number of abstractions and their operating system requirements, along with other communication issues requiring operating system intervention.

11.5.1 Communication Styles

When our applications are composed from multiple, concurrent tasks (e.g., threads, hardware tasks, software tasks, operators), the tasks must often exchange intermediate data in order to solve the entire problem. Specifying this communication can be highly error prone and performance critical. The form in which the communication is specified should match both the natural compute model (see Section 5.1) for the application and the nature of the communication required.

Shared memory

Shared memory is an implicit form of communication motivated by certain implementations where tasks share a common memory pool (Single memory pool subsection of Section 5.1.4) and address space, or share a mapped portion of an address space (Section 11.5.2). Here, the semantics are that each task sees the same image of memory. If one task writes to the image, another should be able to see the values written to the memory. In this way, the memory addresses serve as named locations through which values can be exchanged among tasks.

Uniprocessor operating system developers see shared memory as a particularly efficient way of communicating between tasks. In multithreaded environments where tasks are interleaved in time on the same processor, shared memory segments within the single memory hierarchy allow multiple tasks to share data without an explicit need for data to be copied between the routines. This can minimize the overhead for data communication between tasks. Without caches, shared bus multiprocessing systems with a common main memory would exhibit a similar efficiency. Local caches potentially complicate the picture. However, good architecture and engineering can maintain this abstraction efficiently in the common case, at the cost of additional hardware to support cache coherence.

A reconfigurable computing architecture may nevertheless more closely mirror, at the chip or board level, the organization of a large, distributed memory system.

Here, data may actually need to be copied between distant memories, complicating the shared memory abstraction. The result is both significant hardware overhead to support the model and, often, significant communication time overhead beyond what would be required to move the data between the producer and the consumer. Furthermore, synchronization between shared memory threads/tasks (Section 11.6.1) is a common source of application errors, leading some to question the viability of this model for capturing larger-scale parallelism [21, 36].

Method calls

As the previous section suggested, word-level shared memory is a very low-level form of implicit communication that is prone to synchronization issues. Implementing the abstraction can increase hardware requirements in architectures containing distributed memories. In modern object-oriented systems, particularly when each object may itself be an independent thread, a higher-level communication technique is method calls between objects or operators (see Section 5.1.2). The method call on the object explicitly states the intended destination for the data; further, the object method provides additional semantic information to the receiver about what the data means. As long as object methods are serialized on each object, method invocation can be atomic, providing a natural mechanism for consistent updates to object state. In some cases, method calls can eliminate the need for a hardware task to communicate directly with memory, allowing many lightweight, reconfigurable operators to avoid the expense of a memory interface unit.

When the destination object is running on hardware that is physically distinct from the sending object, the method invocation, and the communication in general, requires data to be routed from the sending to the receiving hardware. This is true even in a shared memory implementation—the method call style simply makes this communication explicit. However, when the objects share the same physical memory, method call communication can still occur through shared memory.

Message passing (discussed in the Message passing subsection of Section 5.2.6) is a form of method call communication, as is remote procedure call [30]. MPI [38] is a well-developed standard for message passing, and reconfigurable computers have been built to interface with standard MPI communications [32]. However, MPI itself is fairly heavyweight, and its overhead may be too high for finer-grained composition of tasks and operators. Lighter-weight message passing designed for on-chip reconfigurable applications has been developed [31], as have remote procedure call interfaces for symmetric use between processors and reconfigurable logic [8].

Streams

While method invocation is an explicit communication mechanism, it is still dynamic and does not provide the OS with advanced warning about which tasks will communicate and when. Further, the actual graph of communication remains implicit in the object call structure. A more explicit form of

communication is to represent the graph structure for task communications and share that information with the operating system. This is similar to the use of pipes or streams in conventional software multi-threading to represent persistent communication links between communicating threads. The reconfigurable computing dataflow models in Section 5.1.3, and the streaming dataflow programming approaches in Chapters 8 and 9 provide some ways to capture these communication graphs. Data-centric compute models (Section 5.1.6) do so as well.

Streams (pipes, channels) are persistent, unidirectional links between tasks (software or hardware) that pass data or control information. Tasks receive available data from one or more input streams and write the results of their computation to one or more output streams [9, 13]. A stream may buffer data in a FIFO manner between the producer and consumer to allow them to run independently of each other and minimize the effects of both reconfiguration and communication latency. Figure 11.4 is an example that shows abstract use of streams (a) and its implementation on a streaming architecture (b). Sections 5.1.3 and 5.2.1 and Chapter 9 present in-depth discussions of streaming models and architectures.

Because the structure of communication (producer–consumer) is explicit, the operating system is able to more easily make intelligent decisions about where to place tasks to promote physical locality, and the scheduler is able to better choose when to run them. For example, if a stream between a producer and consumer is empty or near empty, the scheduler knows that it is more profitable to run the producer than the consumer. A very full stream would imply the opposite.

The persistence of abstract streams allows us to separate the part of communication that specifies the location (source/destination) of data from the part

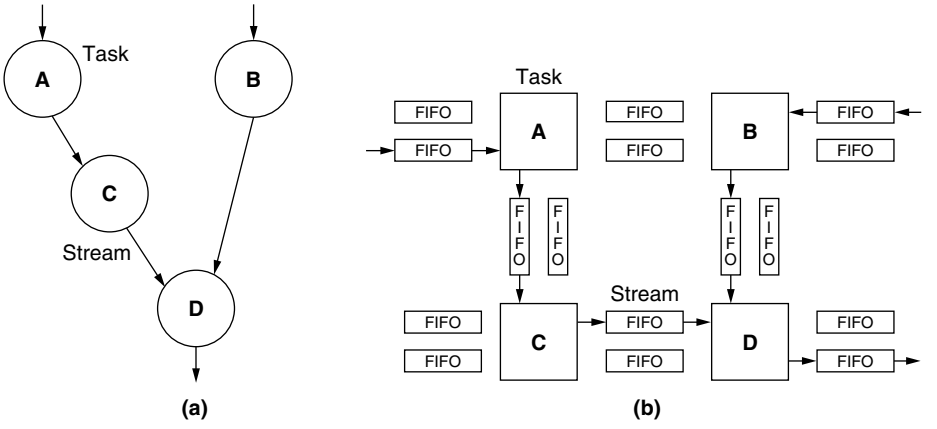

FIGURE 11.4 ■ A stream abstraction defines application dataflow (a); a streaming architecture can implement the streams between tasks using FIFOs (b).

that provides or uses it. For regular communications, this brings the destination specification out of the inner loop of communication, reducing communication overhead. For spatial, reconfigurable datapaths, it allows a stronger correlation between the abstraction and implementation of communication between currently-executing hardware tasks, reducing overhead. The stream can be implemented with simple wires, or a FIFO, between the producer and consumer. Nonetheless, although specifying, allocating, and setting up the stream can be expensive, for heavily used, persistent communications, the long use over time amortizes the cost of stream setup. Short communication sequences or communications to short-lived tasks may not be able to amortize this cost and may be better served with a different communication scheme.

Stream abstraction can be implemented efficiently on a variety of physical communication structures. It can be supported efficiently on a shared memory system with the use of a well-designed and well-tested queue object library that encapsulates the explicit synchronization necessary to implement the stream. Encapsulation is a huge benefit in that it allows one highly trained system programmer to work out a robust locking discipline that can then be used by other programmers with less (or no) experience with synchronization primitives. Stream data can be packed into efficient, longer messages on packet-switched, message-passing systems, or it can be supported by concurrent direct memory access (DMA) data transfers. A message-passing implementation of a stream abstraction can also be extended across the Internet using TCP/IP connections. As noted earlier in this section and elaborated in Chapter 9, when the source and the sink are coresident, the stream can reduce to a direct, configured connection between tasks, requiring minimum hardware and latency overhead during operation.

11.5.2 Virtual Memory

Software applications for general-purpose systems use a virtual memory abstraction, enabled by a combination of hardware and software, to simplify the programming model and to provide isolation (protection) from other processes. Reconfigurable computing systems require this abstraction for the same reasons.

To avoid the complexities of virtual address translation in reconfigurable hardware, the reconfigurable computing system designer may place the burden of memory communication on host processor resources, which already support virtual memory. When the reconfigurable unit is tightly coupled with a processor, it can explicitly share the processor's memory management unit (MMU) [18]. Alternately, the processor can perform memory accesses for a hardware task, feeding data to the task through a dedicated buffer structure [15]. The drawback of using the processor in this fashion is a lack of efficiency. The processor is consigned to acting as an overqualified memory controller, which reduces its availability for parallel computation.

To leverage the processor's address translation capability (including translation lookaside buffer [TLB] miss processing and page fault handling) and at the same time remove the processor from the inner memory access loop, a DMA-style

approach can be used. The processor provides hardware with translated physical addresses for the needed virtual addresses. User hardware should not, however, be able to issue these accesses directly, as it could potentially issue memory requests to other physical addresses outside the task's virtual memory space. An architectural solution to this problem is to add one or more hardware memory address generators that are guaranteed to abide by the virtual memory abstraction. The address generator may require the processor to translate all addresses, or it potentially can combine offsets from the hardware task with a translated page or segment base address to further reduce processor involvement.

Finally, a dedicated hardware MMU can directly translate virtual addresses to physical ones [16, 42]. It maintains its own copy of the TLB for address lookups. TLB misses can be handled either by the hardware MMU itself or by interrupting a processor to walk the page table. In this arrangement, page faults are handled by the operating system, which updates the hardware MMU's TLB based on the result.

11.5.3 I/O

Finally, in addition to communicating with other tasks, a hardware task may need to communicate with system I/O. Libraries abstract the hardware interfaces for the programmer [11] (as discussed in Chapter 8). However, I/O standards are continually evolving and can do so during the lifetime of a given application. The operating system, through I/O device drivers, can support changing I/O standards by providing these libraries in dynamically linked form so that they can be updated and expanded without requiring any changes to the applications in order to use them.

11.5.4 Uncertain Communication Latency

Communication between tasks (and memory) is subject to uncertain latencies for a number of reasons. One common example in many traditional computing systems is the uncertain latency of memory access due to location in the memory hierarchy and memory contention. Reconfigurable computing systems share this problem. However, those that support flexible binding (Section 11.3) are subject to additional sources of uncertainty, as different implementations of a given task have different data rates. Even given the same implementation of a hardware task, its location on hardware can affect the latency of communication between it and other tasks. Depending on the physical implementation of the routing network between physical task locations, some locations may be "closer" than others.

Although this could create variable clock rates depending on task locations, the problem is easily addressed using pipelined interconnect and data presence (discussed in the Data presence subsection of Section 5.2.1 and in Chapter 9). The same set of data presence techniques also support flexible binding where a task implemented in hardware can have a much higher data rate than one implemented in software.

11.6 SYNCHRONIZATION

Reconfigurable computing applications are generally concurrent, executing one or more hardware tasks in parallel along with one or more software tasks. Therefore, they require synchronization between tasks. A number of factors complicate synchronization in reconfigurable computing. First, reconfigurable computing applications can leverage a variety of parallelism types (instruction-level, data-level, task-level, pipeline-level) to a greater degree than software-only applications can, as discussed in Chapter 5. More parallelism exacerbates the already difficult process of concurrent programming [33]. Furthermore, runtime binding and placement can affect communication source/destination locations and task data rate even after program specification and compilation. Given this degree of parallelism and uncertainty, effective synchronization techniques are critical to reconfigurable computing application design and performance.

These effects are mitigated to some extent by the fact that reconfigurable computations and data often use distinct resources with less potential sharing; this can often clarify the synchronization required and permits more coarse-grained resource locking. Depending on the abstraction employed, synchronization may be controlled explicitly by the programmer or implicitly by the operating system or underlying hardware.

11.6.1 Explicit Synchronization

Synchronization between tasks can be performed explicitly through abstractions similar or identical to those used in software-only multi-threaded programming. This approach is particularly appealing in embedded systems, where application designers may have used a shared memory multi-threaded model more widely than the average general-purpose computer programmer would have. As in software-only shared memory applications, constructs such as locks and semaphores can protect access to shared resources to avoid race conditions.

We can impose thread-style interfaces on hardware tasks [4, 7, 42]. The thread interface requests/releases a semaphore and forces hardware to stall or sleep while waiting to acquire one. Memory structures within the hardware must be augmented with a table to hold semaphore information. This has the advantage of hiding details of the hardware task implementation from the communicating thread but at the cost of logic overhead to interface hardware with the shared memory pool that holds the synchronization address.

11.6.2 Implicit Synchronization

Low-level, thread-style synchronization, already prone to design error and debug difficulty, is likely to become even more difficult to implement correctly as the degree of parallelism required to achieve demanded performance increases [21, 36]. Instead, designers could turn to abstractions that provide more explicit parallelism with implicit synchronization.

To efficiently use our reconfigurable resources, we typically provide them with large blocks of data at a time contained in contiguous memory addresses

(e.g., an image frame). Thus, it is natural to give exclusive ownership of a memory block to a hardware task during its execution. By combining this locking with the instantiation semantics for the operation, we can automate locking to prevent the programmer from having to manage it explicitly. This can even be supported by hardware using a scoreboarding technique similar to the ones used to prevent hazards in aggressive processor pipelines [19].

Synchronization is implicit in all forms of dataflow (Chapter 5, Section 5.1.3). The semantics of its operation are based on data arrival, not sequential timing, which makes proper synchronization the job of the compiler, the hardware, and the runtime system rather than the programmer. In streaming dataflow, stream data comes with data presence information (see Section 11.5.1 and Chapter 9). In general dataflow, I-structures allow fine-grained synchronization and concurrent cooperation on common data structures [5].

11.6.3 Deadlock Prevention

Whether synchronization is implicit or explicit, the need for it in a concurrent application presents the unfortunate opportunity for deadlock. Essentially, one or more tasks in the application may not be able to continue because they are waiting on other tasks. When the waiting set forms a cycle, the system will never be able to make forward progress. However, because deadlock can arise only when a task needs exclusive access to multiple resources simultaneously, many hardware tasks will work on a single, coarse-grained set of data at a time, avoiding this issue. Nonetheless, it is common for a hardware task to need multiple resources (e.g., one or more input buffers and an output buffer).

A common method to prevent deadlock is to force tasks to acquire all of their resources in a canonically ordered sequence. This way we avoid deadlock by never creating a cyclic dependence that could lead to it. With implicit and higher-level locking, runtime support mechanisms can provide the ordering guarantee. This demands that we establish a canonical ordering for all resources that might be locked, both in hardware and in memory locations, and use it uniformly throughout the system.

11.7 PROTECTION

Modern computing systems all share a need for protection from processes (intentionally or unintentionally) interfering with one another. This protection is critical for dealing with not only maliciously coded applications but also poorly programmed ones. During the application development process, isolation is critical because it allows designers to test and inspect their implementations. Development is significantly more complicated if bugs can bring down the development system, destroying state information critical to the debugging process. The same need for protection holds for reconfigurable computing systems. The operating system must prevent processes from

using hardware inappropriately or from interfering with or intercepting communication between tasks (hardware or software) of other processes. Some of these responsibilities fall to the scheduler—preventing task resource starvation is one example; others fall to the hardware allocator (which may be part of the scheduler); still others fall to the system's hardware interface.

11.7.1 Hardware Protection

Implementing user tasks as hardware circuits in the reconfigurable fabric introduces a major security flaw unfathomable to the average software user or developer. Depending on the underlying hardware design, a hardware task can cause a short circuit, permanently damaging the computing system. Therefore, either the hardware structure itself must prevent the possibility of short circuits [3, 46] or the operating system must screen user hardware and prohibit any implementations that cannot be proven to be free of short circuits.

Even if an individual task does not cause a short circuit, incorrectly allocating hardware resources to more than one task can create one. That is why the allocation process must physically separate tasks [44]. Figure 11.5 shows a generic FPGA architecture with resources allocated to two different tasks (separated by the heavy dashed line). Wires shown in bold cross the boundaries between tasks, causing potential conflicts. Resources that cross task boundaries can be allocated to no more than one task unless they are part of intertask communication (discussed in the next section). This restriction also prevents maliciously designed tasks from "snooping" communication paths to which they should not have access (also discussed in the next section).

General FPGA structures complicate the task interference problem by having large numbers of extremely flexible routing structures that may span large distances in the hardware. In contrast, some architectures designed specifically for

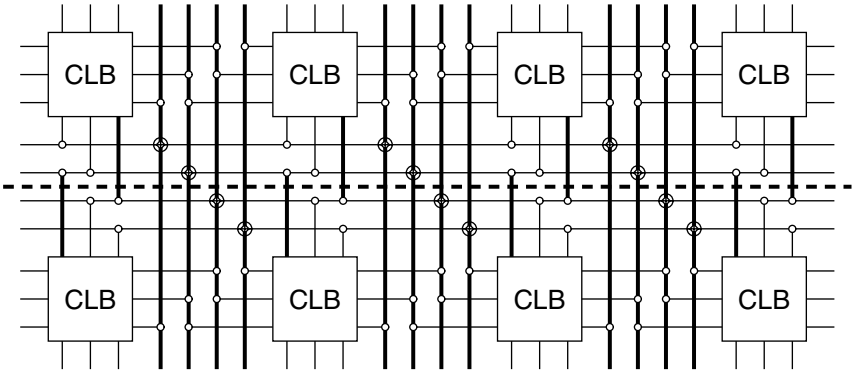

FIGURE 11.5 ■ A generic FPGA architecture may have resources (*bold lines*) that cross the boundary (*dashed line*) between two hardware tasks.

reconfigurable computing, such as SCORE (Chapter 9 and [13]) and PipeRench (Chapter 2, Section 2.1.2, and [17]), are composed of sets of reconfigurable logic (pages/blocks/stages) that are more self-contained, and are the atomic hardware unit for task allocation. Restricted, well-structured connections between these blocks simplify the problem of preventing cross-task interference.

11.7.2 Intertask Communication

As discussed in Section 11.5.2, virtual memory provides each process with a separate address space, preventing one process from accessing the memory space of another. For the same reason, we must provide similar isolation for other forms of communication.

Point-to-point communication, too, can provide isolation if we can guarantee that tasks can only access communication paths owned by their process. The programming model may support this view, but simply trusting it would be equivalent to trusting that compilers will not allow hackers to create viruses. The system (hardware and operating system) must ensure that the isolation model is enforced.

To provide isolation, the system could allow only indirect intertask communication through shared virtual memory [15, 16]. However, this approach can introduce significant communication latencies if both tasks are present in hardware close to one another, but communicate through a relatively distant memory hierarchy acting as intermediary. Safe direct on-hardware intertask communication can be implemented by treating intertask communication routing as special resources that cannot be self-allocated by a hardware task description. Instead, the operating system must allocate these resources when configuring the related tasks onto hardware [13]. By removing user control over allocation of these resources, the isolation the programming model provides is implemented by the operating system. This is much like how only the OS is allowed to manipulate the page tables and TLBs that support the virtual memory abstraction.

11.7.3 Task Configuration Protection

The loading of tasks into hardware must be restricted to the operating system to ensure that the OS has an accurate view of hardware for scheduling/allocation decisions and to enforce hardware and communication protection as discussed previously. Hardware communication paths must therefore be accessible only to OS kernel-level processes. An operating system can isolate task addressability by employing a model akin to virtual memory, where each process can address its own tasks only. Any tables of task information used by the operating system in this case include the process ID as part of the task ID. Any requests for task access are within the user task ID space. Isolation not only prevents processes from triggering the execution, reconfiguring, removing, or altering of tasks from another process, but it also reinforces the abstraction that processes have the hardware to themselves.

11.8 SUMMARY

The primary role of the operating system is to provide abstraction. Abstraction benefits the application designer in the following ways:

- By simplifying the design process to remove the burden of low-level details.
- By allowing the application to run on various hardware platforms and capacities.
- By implementing a virtual machine for each application to prevent interference between them.

This chapter presented the needs, opportunities, benefits, and techniques surrounding the abstraction of reconfigurable resources. It also showed how abstraction affects the application specification process, and discussed the issues involved in implementing these abstractions in the operating system and architecture of the reconfigurable computing system.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, D. Buger. Clock rate versus IPC: The end of the road for conventional microarchitectures. *International Conference on Computer Architecture*, 2000.

[2] A. Ahmadinia, C. Bobda, D. Koch, M. Majer, J. Teich. Task scheduling for heterogeneous reconfigurable computers. *Symposium on Integrated Circuits and System Design*, 2004.

[3] R. Amerson, R. Carter, W. Culbertson, P. Kuekes, G. Snider, L. Albertson. Plasma: An FPGA for million gate systems. *ACM International Symposium on Field-Programmable Gate Arrays*, 1996.

[4] D. Andrews, D. Niehaus, R. Jidin. Implementing the thread programming model on hybrid FPGA/CPU computational components. *Workshop on Embedded Processor Architectures*, *International Symposium on Computer Architecture*, 2004.

[5] Arvind, R. S. Nikhil, K. Pingali. I-Structures: Data structures for parallel computing. *Proceedings of the Workshop on Graph Reduction*, 1986.

[6] G. Brebner. A virtual hardware operating system for the Xilinx XC6200. *International Workshop on Field-Programmable Logic and Applications*, 1996.

[7] G. Brebner. Multithreading for logic-centric systems. *International Conference on Field-Programmable Logic and Applications*, 2002.

[8] M. Budiu, M. Mishra, A. Bharambe, S. C. Goldstein. Peer-to-peer hardware–software interfaces for reconfigurable fabrics. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002.

[9] M. Butts, A. M. Jones, P. Wasson. A structural object programming model, architecture, chip and tools for reconfigurable computing. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2007.

[10] Commodore Business Machines. *Commodore 64: Programmer's Reference Guide*, H. W. Sams, 1982.

[11] C. Chang, J. Wawrzynek, R. W. Brodersen. BEE2: A high-end reconfigurable computing system. *IEEE Design and Test of Computers* 22(2), 2005.

[12] M. Dales. Managing a reconfigurable processor in a general purpose workstation environment. *Design, Automation and Test in Europe*, 2003.
[13] A. DeHon, Y. Markovskiy, E. Caspi, M. Chu, R. Huang, S. Perissakis, L. Pozzi, J. Yeh, J. Wawrzynek. Stream computations organized for reconfigurable execution. *Microprocessors and Microsystems* 30, September 2006.
[14] W. Fu, K. Compton. An execution environment for reconfigurable computing. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005.
[15] W. Fu, K. Compton. A simulation platform for reconfigurable computing research. *International Conference on Field-Programmable Logic and Applications*, August 2006.
[16] P. Garcia, K. Compton. A reconfigurable hardware interface for a modern computing system. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2007.
[17] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, R. Laufer. PipeRench: A coprocessor for streaming multimedia acceleration. *International Symposium on Computer Architecture*, May 1999.
[18] J. R. Hauser. *Augmenting a Microprocessor with Reconfigurable Hardware*, Ph.D. thesis, University of California, Berkeley, 2000.
[19] J. A. Jacob, P. Chow. Memory interfacing and instruction specification for reconfigurable processors. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 1999.
[20] H. Koptez. *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, 1997.
[21] E. Lee. The problem with threads. *Computer* 39(5), May 2006.
[22] B. Levine, H. Schmit. Efficient application representation for HASTE: Hybrid architectures with a single, transformable executable. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003.
[23] Z. Li, K. Compton, S. Hauck. Configuration caching management techniques for reconfigurable computing. *IEEE Symposium on FPGAs for Custom Computing Machines*, 2000.
[24] Z. Li, S. Hauck. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 2002.
[25] J. W. S. Liu. *Real Time Systems*, Prentice-Hall, 2000.
[26] R. Maestre, F. J. Kurdahi, M. Fernández, R. Hermida, N. Bagherzadeh, H. Singh. A framework for reconfigurable computing: Task scheduling and context management. *IEEE Transactions on VLSI* 9(6), December 2001.
[27] Y. Markovskiy, E. Caspi, R. Huang, J. Yeh, M. Chu, J. Wawrzynek, A. DeHon. Analysis of quasi-static scheduling techniques in a virtualized reconfigurable machine. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2002.
[28] G. H. Mealy. The functional structure of OS/360, Part I: Introductory survey. *IBM Systems Journal* 6(1), 1966.
[29] N. Moore, A. Conti, M. Leeser, L. S. King. Writing portable applications that dynamically bind at run time to reconfigurable hardware. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2007.
[30] B. J. Nelson. *Remote Procedure Call*, Xerox Palo Alto Research Center technical report, 1981.
[31] V. Nollet, P. Coene, D. Verkest, S. Vernalde, R. Lauwereins. Designing an operating system for a heterogeneous reconfigurable SoC. *Proceedings of the Reconfigurable Architectures Workshop*, 2003.

[32] A. Patel, C. A. Madill, M. Saldana, C. Comis, R. Pomes, P. Chow. A scalable FPGA-based multiprocessor. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.
[33] S. Qadeer, D. Wu. KISS: Keep It Simple and Sequential. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2004.
[34] J. Rabaey. Reconfigurable processing: The solution to low-power programmable DSP. *Proceedings of ICASSP*, April 1997.
[35] J. Resano, D. Mozos, F. Catthoor. A hybrid prefetch scheduling heuristic to minimize at runtime the reconfiguration overhead of dynamically reconfigurable hardware. *Design, Automation, and Test in Europe*, 2005.
[36] S. Singh. Integrating FPGAs in high-performance computing: Programming models for parallel systems—the programmer's perspective. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2007.
[37] G. Snider, B. Shackleford, R. J. Carter. Attacking the semantic gap between application programming languages and configurable hardware. *International Symposium on Field-Programmable Gate Arrays*, 2001.
[38] M. Snir, W. Gropp. *MPI*: *The Complete Reference*, 2nd ed., MIT Press, 1998.
[39] C. Steiger, H. Walder, M. Platzner. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Transactions on Computers* 53(11), 2004.
[40] A. S. Tanenbaum. *Modern Operating Systems*, Prentice-Hall, 1992.
[41] R. Tessier, W. Burleson. Reconfigurable computing and digital signal processing: A survey. *Journal of VLSI Signal Processing* 28(1–2), 2001.
[42] M. Vuletic, L. Pozzi, P. Hauck. Seamless hardware-software integration in reconfigurable computing systems. *IEEE Design and Test of Computers* 22(2 N), 2005.
[43] H. Walder, M. Platzner. Online scheduling for block-partitioned reconfigurable devices. *Design, Automation and Test in Europe*, 2003.
[44] G. Wigley, D. Kearney. The development of an operating system for reconfigurable computing. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.
[45] M. J. Wirthlin, B. L. Hutchings. A dynamic instruction set computer. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.
[46] Xilinx. *XC6200 FPGA Advanced Product Specification*, June 1996.
[47] B. Ylvisaker, B. Van Essen, C. Ebeling. A type architecture for hybrid micro-parallel computers. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.