CHAPTER 4

Reconfiguration Management

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

The flexibility of reconfigurable devices allows them to be customized to a wide variety of applications. Even individual applications can benefit from reconfigurability by using the hardware to perform different tasks at different times. If not all of an application's configurations fit on the hardware simultaneously, they can be swapped in and out as needed. In some cases, the circuitry implemented on reconfigurable hardware can also be optimized based on specific runtime conditions, further improving system efficiency. The process of reconfiguring the hardware at runtime, whether to accelerate different applications or different parts of an individual application, is (unsurprisingly) called *runtime reconfiguration* (RTR).

Unfortunately, although RTR can increase hardware utilization, it can also introduce significant *reconfiguration overhead*. Reconfiguring the hardware, depending on its capacity and design, can be very time consuming. Modern high-end FPGAs can have tens of millions of configuration points, and writing this information can require on the order of hundreds of milliseconds [3, 54]. In a reconfigurable computing system, where the compute-intensive portions of applications are implemented on reconfigurable hardware, computation and reconfiguration are mutually exclusive operations. Thus, time spent reconfiguring is time lost in terms of application acceleration. Studies estimate that, in some cases, reconfiguration time alone occupies approximately 25 to 98 percent of the total execution time of a reconfigurable computing application [36, 42, 50, 51]. Therefore, management and minimization of reconfiguration overhead to maximize the performance of reconfigurable computing systems is essential.

We first discuss the process of reconfiguration in Section 4.1 and then present different configuration architectures, including those designed specifically to help reduce reconfiguration overhead, in Section 4.2. Section 4.3 discusses the different issues in and approaches to managing the reconfiguration process to minimize reconfiguration overhead and maximize the benefit of hardware acceleration. Section 4.4 focuses on techniques that specifically reduce the configuration transfer time when a reconfiguration is required. Finally, Section 4.5 discusses configuration encryption to maintain intellectual property security in reconfigurable computing systems.

4.1 RECONFIGURATION

In reconfigurable devices, such as field-programmable gate arrays (FPGAs), logic and routing resources are controlled by reprogrammable memory locations, such as SRAM or Flash RAM. Boolean values held in these memory bits control whether certain wires are connected and what functionality is implemented by a particular piece of logic. The process of loading the Boolean values into these memory locations is called *reconfiguration*. A specific sequence of 1s and 0s for particular memory locations in hardware defines a specific circuit and is called a *configuration* for a given hardware task. Runtime reconfiguration therefore involves reconfiguring the device (loading a new set of 1s and 0s) with a different configuration (a specific sequence of 1s and 0s) from the one previously loaded in the reconfigurable hardware (RH). The configurations themselves are created by CAD software based on both the circuit design to be implemented and the architecture of the implementing RH. The architectural information is required for the design tools to know which configuration bits control which resources and what effect a 1 has versus a 0 in each of the configuration bit locations.

Once generated by the CAD tools, configurations are generally stored in a memory structure external to the RH. In some cases, configurations are stored in main memory and a CPU acts as the go-between, transferring them from memory to the RH as needed. In other cases, configurations are stored in a programmable ROM and a *configuration controller* loads the data directly from the ROM in the RH, potentially at the request of a central processing unit (CPU). The configuration controller and the ROM may be incorporated into the same device, such as the specialized configuration controllers marketed by various FPGA companies [3, 55], or they may be part of a user-designed custom device. Figure 4.1 shows a block diagram of a system using a configuration controller triggered by a CPU to reconfigure the RH (in this case, an FPGA). The configuration controller essentially implements a finite-state machine (FSM) that, based on the configuration requested by the CPU, generates the sequence of addresses needed to read the appropriate data sequence for that configuration out of the ROM.

4.2 CONFIGURATION ARCHITECTURES

A configuration architecture is the underlying physical circuitry that loads configuration data during reconfiguration, and holds it at the correct locations. Configuration architectures can range from simple serial shift chains, as discussed in the next section, to addressable structures that can manipulate configuration information after it is loaded. Some researchers have developed methods to emulate more complex configuration architectures on existing commercial designs, using a combination of hardware and software to provide advanced configuration functionalities. These approaches are discussed in Section 4.3.4.

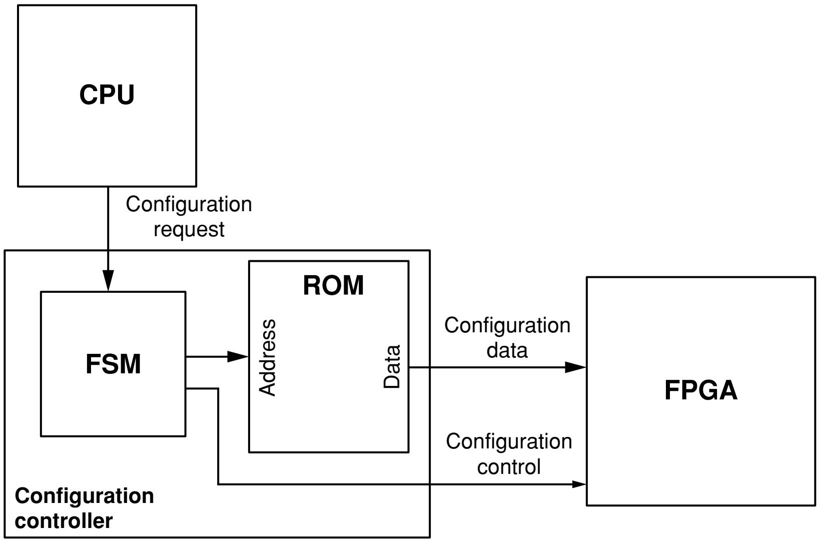

FIGURE 4.1 ■ Configuration data can be transferred to an FPGA by a specialized configuration controller containing nonvolatile ROM memory; the reconfiguration process can be triggered by a CPU.

4.2.1 Single-context

The single-context FPGA has been the most common choice in commercial designs, though there are exceptions. In this type of FPGA, configuration information is loaded into the programmable array through a serial shift chain, as shown in Figure 4.2.

Internally, the configuration architecture may actually be addressable, similar to a standard RAM device or the partially reconfigurable designs discussed in Section 4.2.3, but this would be an implementation detail hidden from the FPGA user. Addressable configuration architectures generally require fewer transistors per SRAM cell than serially programmed architectures, reducing the area required for configuration memory. In this case, an internal-state machine would control writing serially received data to locations in the array.

The Xilinx Virtex family of FPGAs have addressable configuration locations, but have a single-context configuration mode [54]. In these FPGAs, configuration data is divided up into addressable blocks called "frames," each of which corresponds to part of a column of reconfigurable resources. During reconfiguration, the configuration data is shifted into the frame data input register (FDRI) and from there written to a configuration memory location specified by the frame address register (FAR). For single-context configuration mode, this address starts at 0 and is automatically incremented each time a new frame is loaded. This allows the device to appear externally as a single-context device despite the addressability of the configuration information.

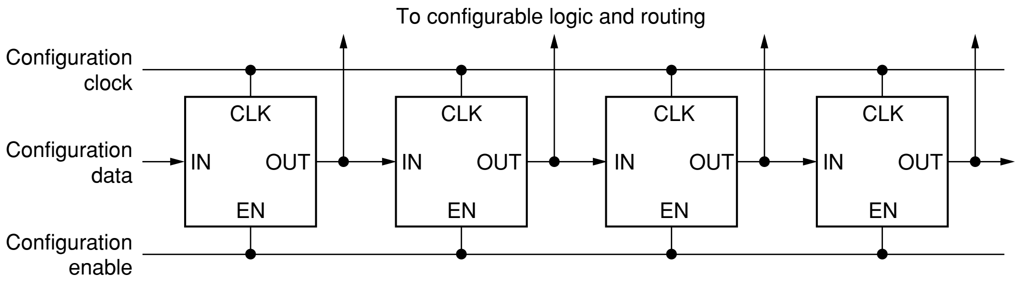

FIGURE 4.2 ■ Serially programmed FPGAs shift in configuration data. Each cell shown contains one SRAM bit of programming data. The clock controls shifting during configuration.

The benefit of serially programmed devices is that they require few pins for configuration, potentially simplifying board-level design. However, the entire chip must be reprogrammed for any change to the configuration data because the data cannot be selectively "reused" on the chip. For example, a large part of the structure of an encryption application may be independent of the chosen key, with only a relatively small portion optimized on a per-key basis. Ideally, only the key-dependent parts are reconfigured and the key-independent parts remain untouched when the key changes. However, a single-context design requires all configuration data to be rewritten during configuration, even if it is with the same values. A relatively minor change to the configuration data becomes a full reconfiguration process, replete with the associated delays.

The number of configuration cycles can be somewhat reduced in single-context devices by widening the configuration path. The Altera Stratix-II [3] and the Xilinx Virtex-II [54] receive either a single bit or a byte of configuration information per configuration clock cycle. The designer then chooses between the two modes by weighing the board-level design impact against the performance impact. As the larger Stratix II devices currently require more than 4MB of configuration data, with a maximum configuration clock speed of 100 MHz, the ability to configure in eight times fewer cycles can be significant. Newer Xilinx devices, such as the Virtex-5, allow a configuration data bus up to 32 bits wide [55].

4.2.2 Multi-context

For RTR systems, the overhead of serial programming may be prohibitive. An attractive alternative may be to provide storage in the device for multiple configurations simultaneously, facilitating configuration prefetching and fast reconfiguration. A *multi-context* device (sometimes called "time-multiplexed") contains multiple planes (contexts) of configuration data. Each configuration point of the device is controlled by a multiplexer that chooses between the context planes. Two configuration points for a 4-context device are shown in Figure 4.3. Several time-multiplexed FPGA architectures have been proposed, including Time-Multiplexed [47], DPGA [17], Dharma [11], and Morphosys [45].

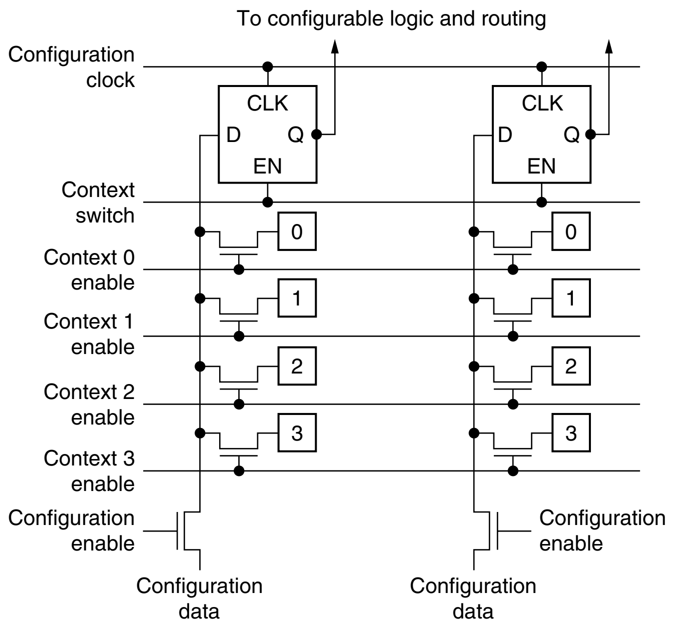

FIGURE 4.3 ■ Two multi-contexted configuration bits of a 4-context device.

Multi-context devices have two main benefits over single-context devices. First, they permit background loading of configuration data during circuit operation, overlapping computation with reconfiguration. Second, they can switch between stored configurations quickly—some in a single clock cycle—dramatically reducing reconfiguration overhead if the next configuration is present in one of the alternate contexts. However, if the next needed configuration is not present, there is still a significant penalty while the data is loaded. For that reason, either all needed contexts must fit in the available hardware or some control must determine when contexts should be loaded in order to minimize the number of wasted cycles stalling while reconfiguration completes. This type of control is discussed in Section 4.3.2.

One of the drawbacks of multi-contexted architectures is that the additional configuration data and required multiplexing occupies valuable area that could otherwise be used for logic or routing. Therefore, although multi-contexting can facilitate the use of an FPGA as virtual hardware, the *physical* capacity of a multi-contexted FPGA device is less than that of a single-context device of the same area. For example, a 4-context device has only 80 percent of the "active area" (simultaneously usable logic/routing resources) that a single-context device occupying the same fixed silicon area has [17]. A multi-context device limited to one active and one inactive context (a single SRAM plus a flip-flop) would have the advantages of background loading and fast context switching coupled with a lower area overhead, but it may not be appropriate if several different contexts are frequently reused.

Another drawback of multi-contexted devices is a direct consequence of its ability to perform a reconfiguration of the full device in a single cycle: spikes in dynamic power consumption. All configuration points are loaded from context memory simultaneously, and potentially the majority of configuration locations may be changed from 0 to 1 or vice versa. Switching many locations in a single cycle results in a significant momentary increase in dynamic power, which may violate system power constraints.

Finally, if any state-storing component of the FPGA is not connected to the configuration information, as may be true for flip-flops, its state will not be restored when switching back to the previous context. However, this issue can also be seen as a feature because it facilitates communication between configurations in other contexts by leaving partial results in place across configurations [27].

4.2.3 Partially Reconfigurable

Because not all configurations require the entire chip area, we might reduce reconfiguration time if we reloaded data only to those areas that actually must change. In partially reconfigurable devices, the configuration memory is addressable, similar to traditional RAM structures. If configurations are smaller than the full device, partial reconfiguration can decrease reconfiguration time by limiting reconfiguration to the resources used by a given configuration and, therefore, the amount of configuration data to transfer. Partial reconfiguration can also allow multiple independent configurations to be swapped in and out of hardware independently, as one configuration can be selectively replaced on the chip while another is left intact. Furthermore, we can leverage the addressability to modify only part of a configuration already located on the chip if some of its structure matches a new configuration that we wish to load. For example, in an encryption circuit the bulk of the configuration may remain the same when the key is changed, and only a few resources may need to change based on the new key value. Partial reconfiguration can allow the system to reconfigure only those changed resources instead of the full circuit.

The Xilinx 6200 FPGA [53] was an early partially reconfigurable device where each logic block could be programmed individually. It therefore became a platform for a great deal of study of configuration architectures and RTR. Current partially reconfigurable commercial FPGAs include the Atmel AT40K [5] and the Xilinx Virtex FPGA family [54, 55]. The Virtex series is more coarsely reconfigurable than the 6200. Instead of addressing each logic block independently, it reconfigures logic blocks in groups called frames. In the Virtex-II, a frame corresponds to part of a full column of resources and the size of the frame increases with the number of logic block rows in the device. In the Virtex 5, frames are a fixed size of 41 32-bit words (regardless of device size) that represent a partial column of resources.

Although partially reconfigurable designs provide a great deal more flexibility for RTR systems, they can still stuffer from potential problems. First, if configurations occupy large areas of the device, the time saved transmitting configuration data may be outweighed by the time spent transmitting configuration addresses.

In this case, a serially programmed FPGA may be more appropriate. Second, and more critical to RTR systems, partial configurations are generally fixed to specific locations on the device. If two independent configurations are implemented in overlapping hardware locations, they cannot operate simultaneously. One method of mitigating this issue is to view configuration placement as a three-dimensional floorplanning problem, with the third dimension representing time [6]. Configurations then occupy some three-dimensional volume of space based on physical location and time of use, allowing the floorplanner to determine the best two-dimensional placement to avoid time-related (three-dimensional) conflicts. Unfortunately, this technique cannot guarantee nonoverlapping configurations if the full configuration sequence is not known at compile time—a major problem in multitasking systems. The next section discusses advanced configuration architectures that eliminate configuration placement conflicts.

4.2.4 Relocation and Defragmentation

As previously discussed, conflicts between configuration locations can limit the effectiveness of partially reconfigurable architectures. To remove these conflicts, configurations should not be associated with fixed device locations. Relocation is a technique permitting configurations to be moved to different compatible device locations within the array, based where free area is available. Figure 4.4(a) shows a device loaded with configurations A, B, and C in sequence, each assigned to a free area. Figure 4.4(b) shows configurations A and B removed, and configuration D relocated and programmed onto the array.

The composition of the reconfigurable hardware can complicate this process in three critical ways. First, if the device's logic or routing is heterogeneous, relocation becomes less flexible, or even impossible, as a configuration may require resources located in only one or a few array locations. For example, in devices with hierarchical routing, different routing connections are available at different locations in the array. However, if heterogeneity is restricted to a repeating pattern, configurations can be relocated distances corresponding to some multiple of the distance of the repeat. To the relocated configuration, resources will be located in the same relative position as in the original placement.

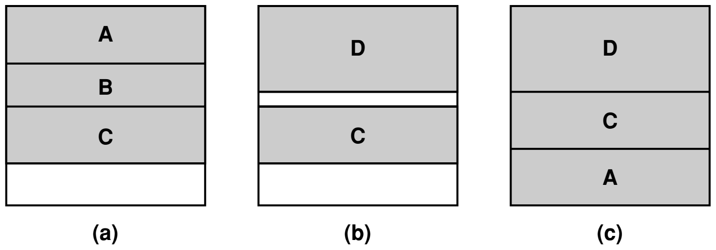

FIGURE 4.4 ■ Three configurations have been programmed on the hardware (a). In (b), A and B have been removed, and D has been relocated/configured to an available area, causing fragmentation. Defragmentation relocates configuration C to make room for configuration A when it is again needed, this time to a new location in the array (c).

Second, the external pin connections to the reconfigurable hardware fabric are fixed either at fabrication (reconfigurable hardware cores in a system-on-a-chip) or at board-level design (discrete FPGA components). If a configuration is relocated, its connections to the required I/O pins must be rerouted to maintain the proper connections. One solution to this problem is to use a communication network that itself has fixed pin connections but provides internal interfaces at multiple array locations to allow configurations to have the same communication connections regardless of position [14, 50]. This type of structure is known as *virtualized I/O* and can in some cases be emulated by using reconfigurable resources to implement a static communication structure and including the communication interfaces in individual dynamic configurations [7]. However, configurations must still be relocated such that they can still connect to the communication bus.

Third, a two-dimensional architecture can exacerbate the previous two problems, but particularly complicating virtualized I/O. If a configuration can be relocated both horizontally and vertically, the virtualized I/O must potentially distribute signals to all locations in the array. Furthermore, a two-dimensional architecture increases the possibilities for relocation, as we can consider not only configuration shifting but also rotation, which requires manipulating configuration information related to routing [14]. More relocation possibilities leads to a more complex relocation process and possibly increased configuration overhead.

A partially reconfigurable architecture designed specifically with relocation support should therefore require a homogeneous logic architecture, a bus-based communication structure, and a one-dimensional organization to simplify the relocation process [31, 50]. The one-dimensional architecture means that a configuration must use complete rows, even if it only needs a portion of a row. As device sizes increase, using rows as atomic reconfiguration units may become inefficient. Instead, the fabric can be split into multiple one-dimensional fabrics to retain the relocation benefits while preserving a reasonably sized atomic unit. The Virtex-5 device uses this approach [55].

One of the architectures designed for relocation [14] uses a "staging area" equivalent in size to one row of configuration data, which is similar in approach to the column-wise frame-based configuration method of the Xilinx Virtex family introduced in Section 4.2.1 and discussed in Section 4.2.3 [54]. The staging area is filled one configuration word at a time; then the entire row of data is simultaneously written to the architecture at a location computed with a base address of the top row of the configuration combined with an offset indicating the position of the current row relative to the top configuration row. The choice of the base location can be made by a special circuit that monitors empty locations on the hardware, or by software. When combined with the proper software as described in Section 4.3.2, this configuration architecture has been shown to reduce reconfiguration overhead by 85 percent over a single-context device [31].

Even if an architecture allows relocation, fragmentation of the usable resources can decrease its effectiveness. Like memory fragmentation, swapping configurations in and out of different places in the hardware can result in a situation where various locations in the array may be unused, but there may not be enough contiguous space available to load a configuration. In this case, if the configurations can be defragmented, the new configuration can be loaded into the array without having to remove any of the configurations already on the device. Figure 4.4(b) shows an example of an array that has become fragmented, and Figure 4.4(c) shows how defragmentation can allow a configuration to be configured without having to remove an existing one. A simple approach to this problem is to remove all configurations, then reconfigure the array with the removed ones, this time relocating them to contiguous locations to eliminate fragmentation. However, this process involves significant communication overhead between fabric and configuration memory. Alternately, the reconfigurable hardware can move configurations internally, avoiding the need to communicate with configuration memory. The R/D FPGA [14, 31] provides both relocation and defragmentation ability, which together provide a 90 percent reduction in reconfiguration overhead compared to a single-context FPGA.

A configuration controller for one-dimensional hardware, such as the R/D FPGA, that specifically supports relocation and defragmentation may simply need to keep track of occupied and unoccupied locations, or request this information as needed from the hardware itself. The controller can determine locations for incoming configurations using a first-fit or best-fit method, similar to general memory allocation [7, 14]. Defragmentation, which is easy for the one-dimensional case, can be triggered when sufficient free area is available but is broken up into fragments too small to fit an incoming configuration. If there is insufficient free area, one or more configurations can be removed to make room, as described in Section 4.3.2.

4.2.5 Pipeline Reconfigurable

Pipeline reconfigurable arrays use a series of physical pipeline stages to implement the virtual pipeline stages of configurations. A virtual pipeline stage can be relocated to any physical pipeline stage, and the number of virtual stages is generally not constrained by the number of physical stages. The most well-known pipeline reconfigurable architecture is PipeRench [19], which is designed to implement deeply pipelined configurations, subdivided into a set of virtual pipeline stages. At runtime, the virtual pipeline stages are assigned to physical pipeline stage computation units. These units are arranged in a unidirectional ring, as shown in Figure 4.5(a). Although pipeline stages may be implemented in different physical locations over time, the virtual pipeline *appears* fixed to its own pipeline stages, with each stage receiving input from its predecessor and generating output to its successor. PipeRench permits pipeline stages to be configured in a single cycle to speed execution.

Pipeline reconfiguration eliminates many of the difficulties of using reconfigurable hardware as virtual hardware, but places restrictions on the circuits that can be implemented as information can only propagate forward through the pipeline stages, and any feedback connections must be completely contained within a single stage. Figure 4.5(b) shows a 4-stage virtual pipeline implemented on a 3-stage physical architecture.

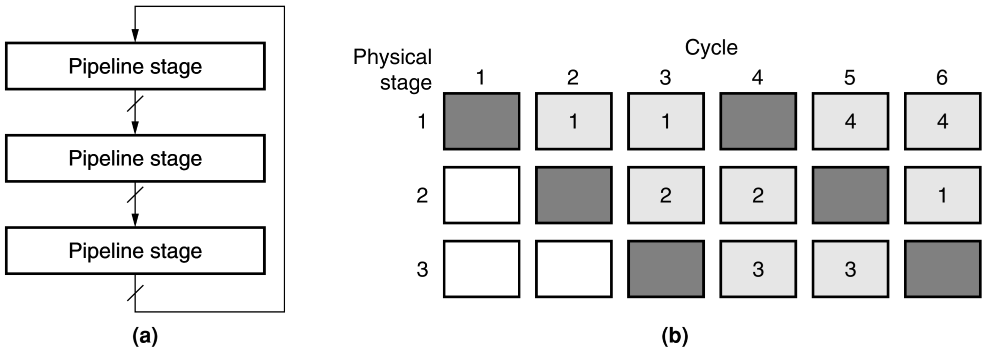

FIGURE 4.5 ■ A pipeline reconfigurable architecture with three physical stages (a). A 3-stage physical pipeline implementing a 4-stage virtual pipeline (b). Numbers within physical pipeline stages indicate the implemented virtual pipeline stage. Shaded stages are reconfiguring for the given cycle.

4.2.6 Block Reconfigurable

Block reconfigurable arrays can share characteristics with any of the previously described configuration architectures. However, rather than providing one large reconfigurable fabric, they are made up of multiple discrete blocks that can be used independently. For these purposes, "block" should not be confused with "logic block" in an FPGA. In this case each independent block can contain many logic resources. An individual configuration may occupy one or more blocks, but blocks may not be subdivided between configurations. Blocks are connected either through a crossbar structure [39] or a bus/network [10], as shown in Figure 4.6. Although this would seem to describe any architecture formed from multiple connected FPGAs or FPGA cores, block reconfigurable devices have the ability to relocate configurations to different blocks at runtime. For this reason, the blocks of reconfigurable logic in this style of architecture have also been referred to as "swappable logic units" (SLU) [55]. In the SLU architecture, a block reconfigurable design is implemented as an abstraction layer on top of a partially reconfigurable architecture to facilitate runtime relocation.

The SCORE reconfigurable architecture model [10] is a block reconfigurable design where the reconfigurable blocks are referred to as "pages" to evoke a virtual memory view of the reconfigurable hardware. Any virtual page can be implemented on any physical page, and computation pages are loaded as needed. Once configured, pages communicate with one another using datastreams over a scalable hierarchical network.

A heterogeneous multiprocessor may fit the block reconfigurable model, provided multiple blocks of reconfigurable hardware are present and configurations can be relocated between the blocks for computational flexibility. These architectures may contain a single communication network used by the configurable blocks and other resources such as microprocessors and custom circuitry. Although the Pleiades reconfigurable architecture [1] has some of these features (a heterogeneous multiprocessor with multiple reconfigurable blocks),

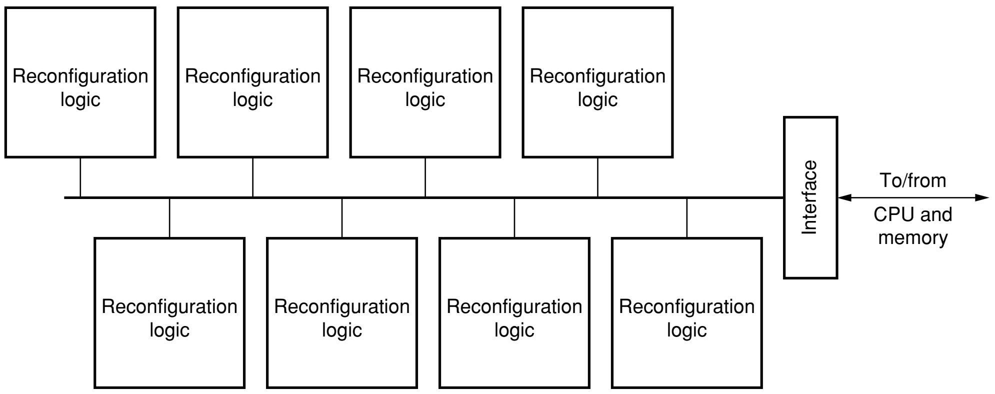

FIGURE 4.6 ■ In a block reconfigurable device, configurations can be relocated to any of the interconnected and equivalent blocks of reconfigurable logic.

computations are preassigned to specific resources, violating one of the requirements of the block reconfigurable category.

4.2.7 Summary

This section presented a variety of configuration architectures, each optimized for a different type of reconfiguration. The single-context device is the simplest in terms of configuration process and interface, and it is the most popular for current commercial devices. Partial reconfiguration, which allows reconfiguration of parts of the device (leaving the rest untouched), can reduce the amount of configuration data that must be transferred but is hampered by configuration placement conflicts. Partially reconfigurable designs augmented with relocation and defragmentation, as well as block reconfigurable designs, avoid this issue by allowing configurations to be placed at different locations from the ones originally assigned. Likewise, pipeline reconfigurable devices allow pipeline stages to be relocated but prohibit interstage feedback connections. Finally, multicontexted devices provide a method for single-cycle device reconfiguration but at the cost of decreased computation resources for a given area and a dramatic increase in power consumption during context changes.

The more advanced reconfiguration architectures, such as relocation, defragmentation, and multi-contexting, have been popular for some time in the research community as tools essential for effective reconfigurable computing systems. However, such devices have not yet gained a significant market foothold because of the limited demand for fast reconfiguration capabilities. Instead, most FPGAs are currently used as drop-in ASIC replacements or as infrequently reconfigured hardware modified only for firmware updates. To provide devices at a competitive cost, most FPGA vendors forgo the more innovated configuration architectures in favor of a simpler single-context design. Although Xilinx, one of the most prominent FPGA vendors, offers partial reconfiguration in its Virtex families, design support is still somewhat limited, relocation and defragmentation are not supported, and a single-context interface is still provided to cater to users who do not require partial reconfiguration. Even so, as reconfigurable computing becomes a more common practice, spurred perhaps by the difficulty of continued clock speed increases for general-purpose processors, demand for innovative configuration architectures will increase in order to maximize the benefits of reconfigurable computing.

4.3 MANAGING THE RECONFIGURATION PROCESS

Reconfigurable computing systems swap configurations in and out of hardware at runtime, a process controlled by software, hardware, or a combination of both. Although a system can simply load a configuration whenever it is needed, and unload it when hardware execution is complete, this can cause a significant reconfiguration overhead: while the configuration is loading, the controlling application or thread cannot compute. Also, if the hardware is currently in use by another thread or process, the requesting application or thread must wait until the hardware is idle or until enough area is free to even begin the reconfiguration process, leading to further stalling. Ideally, configurations are loaded in advance of when they are needed and those likely to be reused in the near future should be cached on the hardware.

The following sections discuss several aspects of reconfiguration control, including choosing the configurations to load, and when and where on the hardware to load them.

4.3.1 Configuration Grouping

Single-context and multi-context FPGAs may have more resources available at once than are usable by a single configuration. Reconfiguration overhead can be reduced by grouping configurations that are likely to be used one after another into a single larger configuration. Algorithms proposed to perform this grouping include simulated annealing and a clustering approach [31]. They examine the overall application control flow to predict configurations that should be grouped together. The loading of a grouped configuration involves not only the currently needed configuration but also those most likely to be used after. Therefore, if the next configuration requested is already present on the device, no reconfiguration is necessary, reducing reconfiguration overhead. With configuration grouping, a configuration will appear in at least one group, and possibly several, depending on application behavior and the configuration's relationship to other configurations.

This approach is primarily appropriate for single-application systems, as configuration grouping is a compile-time operation. However, it could also be used in a multitasking system with a multi-context device. In this case, the configuration grouping would still be performed at compile time for individual applications, and the choice of which configuration groups to load and when would be a runtime operation, as described in the next section.

4.3.2 Configuration Caching

In a single-context device, the loading of one configuration overwrites all configuration data in the FPGA. Thus, context grouping implicitly decides what operations will coexist within the device at any point. In a multi-context or partially reconfigurable architecture, reconfiguration only overwrites a portion of the configuration data, allowing other configurations to be retained elsewhere. With configuration caching, the goal is to keep configurations on the hardware if they are likely to be reused in the near future. If there is enough free area on the device to fit a requested configuration, it is simply loaded, but if there is insufficient space, the configuration controller must select one or more "victim" configurations to remove from the hardware to free the required area. This process is simplified from the point of view of the controller if the device does not support relocation, as the victim configurations are simply any that overlap with the incoming one. However, this will generally result in a high reconfiguration overhead, as the removed configurations could be needed again in the near future, requiring another reconfiguration.

If the device supports relocation and defragmentation, or multiple contexts, the controller may have a variety of potential victims to choose from that will free the needed area. In some cases, general caching approaches may be used. These approaches assume a fixed-sized data block. However, in a partially reconfigurable device the size of the block to load can vary because configurations can each use differing amounts of resources. The caching algorithm must therefore consider the impact of variable-sized blocks.

One algorithm uses a penalty-based approach that considers both the configuration's size and how recently it was used [31]. When a configuration is first loaded, its "credit" is set to its size. When one or more configurations must be removed to make room for an incoming one, the configuration with the lowest credit is chosen, and the credit values of the remaining configurations are lowered by the credit value of the removed one. For the R/D FPGA design [14], penalty-based caching consistently results in a lower reconfiguration overhead than a simple least recently used (LRU) approach and 90 percent less overhead than a single-context configuration architecture. A configuration controller for a multi-context device must select which context to overwrite when a new context not already in the device is requested [14]. Because each context is the same size, general caching techniques, such as LRU, have been used.

4.3.3 Configuration Scheduling

Configurations can be loaded simply as they are requested, but this may result in significant overhead if the software stalls while waiting for reconfiguration to complete [50]. If instead the system can request configurations in advance of when they are needed, a process called *prefetching*, reconfiguration may proceed concurrent with software execution until the hardware is actually required. The challenge, however, is to ensure that prefetched configurations will not be ejected from the hardware by other prefetching operations before they can be used. For example, Figure 4.7 shows a flow graph for an application containing both

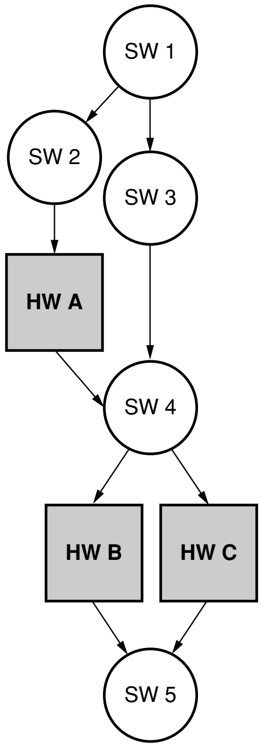

FIGURE 4.7 ■ An example reconfigurable computing application flow graph, containing both hardware and software components.

hardware and software components. Configuration A can safely begin loading at the beginning of the flow graph, provided that the application represented by the flow graph is the only one using the reconfigurable hardware. On the other hand, after the first branch rejoins at software block 4, it is unclear whether configuration B or configuration C will be needed next. If both potential branches have equal probability, the next configuration should not be loaded until after program flow determines the correct branch.

For static scheduling, prefetching commands may be inserted by the compiler based on static analysis of the application flow graph [23], and have been shown to reduce reconfiguration overhead up to a factor of 2. A more dynamic approach uses a Markov model to predict the next configuration that will be needed for a partially reconfigurable architecture with relocation and defragmentation [33]. Combining this approach with configuration caching results in a reconfiguration overhead reduction of a factor of 2 over configuration caching alone. Adding compiler "hints" to dynamic prediction achieves still better results.

Some dynamic approaches use the dataflow graph to determine when a given configuration is valid for execution [37, 39]. In these cases, nodes of the flow graph may be scheduled only if their ancestors have completed execution. This approach works even if multiple applications are executing concurrently in the system and also works in systems implementing hardware tasks as independent "hardware threads" [4, 43].

Other approaches do not consider the actual flow graphs of applications, but instead use system status and current resource demand to allocate reconfigurable hardware to different configurations over time. Window-based scheduling periodically chooses the configurations to be implemented in hardware for the next "window" of time. This approach treats scheduling as a series of static problems yet still accommodates dynamic system behavior. One window-based scheduler uses a multi-constraint knapsack approach to choose configurations providing the best benefit (speedup) to the system as a whole based on configuration requests in the past window period. This technique was shown to increase overall *system* throughput by at least 20 percent relative to a processor without reconfigurable hardware [57].

In true multitasking systems load may not be consistent, with demand for the reconfigurable resources varying over time. This has led to more complex scheduling techniques that also consider modifying configurations based on available resources to take advantage of numerous resources when possible or to fit in limited resources when necessary [37, 40, 41, 57]. Another possibility is to permit a software alternative for configurations to avoid stalls if the hardware resources are in high demand [16, 34, 41, 57]. This approach allows dynamic binding of computations to hardware or software, where only the most beneficial configurations are actually implemented in hardware. Real-time systems similarly must choose tasks at runtime for hardware implementation based on real-time requirements (task priority, arrival and execution time, and deadlines), rejecting remaining tasks to software or possibly dropping them entirely [46].

4.3.4 Software-based Relocation and Defragmentation

Systems that do not support relocation and defragmentation at the configuration architecture level may support it at the software level to gain some of the associated benefits. However, this can be computationally intense for two-dimensional architectures. Finding a possible location for an arbitrarily shaped configuration can require an exhaustive search, which may incur a greater overhead penalty than the configuration penalty it seeks to avoid. Restricting configurations to rectangular shapes simplifies the process somewhat, though it is still a two-dimensional bin-packing problem. One approach to solving this problem is to maintain a list of empty spaces in the device and search it whenever a new configuration is to be loaded [6, 21, 48]. In either case, when the controller removes a configuration from the hardware, it can update the list based on the freed area. The "best" empty location to implement the incoming configuration can be chosen based on algorithms similar to one-dimensional packing, such as first-fit or best-fit.

When there are no empty locations that can fit the incoming configuration, the configuration controller can defragment the hardware to consolidate empty space, or remove an existing configuration. Like two-dimensional relocation, two-dimensional defragmentation is very complex. It can be implemented by removing all configurations from the hardware and then successively reloading each one using one of the two-dimensional relocation techniques described previously. Alternately, a reconfiguration controller can use a technique specifically designed for two-dimensional defragmentation that rearranges only a subset of configurations, and dynamically schedules their movements in an effort to minimize disruption of those in execution [18].

A critical problem in supporting relocation, whether for the one-dimensional or the two-dimensional case, is rerouting the connections between a relocated configuration and the (nonrelocated) I/O pins. As discussed in Section 4.2.4, a virtualized I/O structure simplifies this problem, though virtualized I/O for two-dimensional architectures may be infeasibly large. However, if the architecture does not have virtualized I/O, either these signals must be rerouted at runtime [49] or the configurations must be modified to emulate virtualized I/O by having a specific movable interface to a nonrelocatable communications structure [7].

4.3.5 Context Switching

Unfortunately, some of the same terminology in the reconfigurable computing area is used to refer to different concepts. In this section, "context switch" does not refer to switching between planes of configuration data in a multi-context device. Instead, it refers to the suspend/resume behavior of processors (and potentially their associated reconfigurable logic) when multitasking. A few studies have discussed supporting suspend/resume of hardware operations as a way to support hardware multitasking [24, 44]. In these systems, long-running configurations may be interrupted to allow other configurations to proceed, and later be resumed to complete computation. Although the configuration state can be resumed by reloading the required configuration, the flip-flop values and the values stored in embedded RAM blocks are not necessarily part of the configuration, and therefore may require additional steps to save their state.

Reconfigurable hardware context switches may mirror processor context switches to facilitate hardware control by ensuring that the "owning" process is active and ready to receive results. The host processor may stall or wait while the reconfigurable hardware is active [43], or it may continue with parallel operations that are not dependent on the hardware's results [1, 24, 43].

4.4 REDUCING CONFIGURATION TRANSFER TIME

The various techniques described previously can reduce the number of times we have to reconfigure the hardware, or attempt to hide the configuration latency, but the actual time required to transfer a given configuration can also be reduced. One hardware-based technique already discussed in Section 4.2.3, partial reconfiguration, permits configuring only those parts of the hardware that are needed. The remainder of the chip does not need to be configured, and therefore configuration data for these other areas does not need to be transferred. The next few sections present a number of other methods used to reduce the configuration transfer time, in most cases by reducing the amount of data transferred.

4.4.1 Architectural Approaches

The design of the reconfigurable architecture itself can affect the time required to configure it. For example, a coarse-grained architecture containing primarily fixed functional units will generally require fewer configuration bits for the same functionality than does a fine-grained LUT-based architecture [25]. Another architectural design feature that can impact reconfiguration times is the width of the configuration path. Section 4.2.1 discussed how a serially programmed FPGA can be programmed 8× faster if configuration data is loaded a byte per cycle instead of a bit per cycle. In cases where the reconfigurable hardware is located on the same chip as the configuration memory, a very wide path between them may be possible, drastically reducing reconfiguration time. For example, the R/D architecture [14] can have a wide enough path to an on-chip configuration cache to allow the entire staging area to be loaded in a single cycle.

4.4.2 Configuration Compression

Compression is a widely used method in general-purpose computing and networking to reduce data transfer times by reducing the number of bits transferred. Compression can also reduce the amount of configuration data transmitted to reconfigurable hardware, leading to a corresponding decrease in reconfiguration time. The first proposed configuration compression technique [22] targeted the Xilinx 6200-series FPGA [53], which, as discussed in Section 4.2.3, is partially reconfigurable at a very fine-grained level, addressing individual logic cells by their row and column. The 6200 includes two "wildcard registers," equal in bit width to the row and column addresses, which act as masks on the configuration addresses. This allows one piece of configuration data to be written to more than one location. Essentially, 0s in the wildcard register retain the configuration address bits for those locations, whereas 1s indicate that all possible combinations of values in those specific locations should be addressed. By treating wildcard register value generation as a logic minimization problem, configuration data is compressed by an average factor of four for the Xilinx 6200 [22].

An expansion of these efforts exploits the fact that not all configuration bits in a logic cell are used by all configurations [30]. In many cases, a number of bits in a logic cell configuration can be considered "don't-care" values and can be programmed either with a 1 or a 0 without affecting the configuration's functionality. This allows configuration data to be manipulated to increase the achievable compression rates by about a factor of 2. Although the wildcarding and don't-care approaches are effective, they are specific to a discontinued architecture. More recent studies [15, 32] examine the use of a variety of standard compression techniques that achieve up to a compression factor of 4 on more modern architectures.

Configuration compression is not merely an academic pursuit. Both Altera's and Xilinx's design tools can generate compressed configurations [3, 55]. The compressed configurations are stored in a separate configuration controller that decompresses them as they are sent to the FPGA. However, this form of compression only reduces configuration storage requirements and does not decrease the size of configuration data sent to the FPGA. Compressed configurations can, however, be loaded directly onto Stratix-II devices in some configuration modes, and decompressed on the FPGA itself.

4.4.3 Configuration Data Reuse

At times, only a portion of a configuration must be updated, such as the key-specific hardware in an encryption configuration. Rather than resend the full configuration information, a partially reconfigurable device allows just the changed portions to be sent. Circuits can be designed specifically to use partial reconfiguration to customize them based on constant values not known until runtime [58]. However, even less directly related configurations may also have configuration data in common. Certain computation or communication patterns may be common to several configurations, such as the use of adder structures, emphasis on near-neighbor routing instead of long-distance routing, and the like [20]. Similarly, there may be "default" values for configuration bits for unused resources, and two configurations may have used as well as unused resources in common. These commonalities can decrease the amount of "new" configuration data required to implement the next configuration, particularly if configuration data reuse is a factor in the design of the configurations. The degree of similarity is increased with a decrease in the granularity of reconfiguration (there are fewer ways for small sets of bits to differ than for large sets to differ) and can result in a decrease in configuration data by approximately 35 to 40 percent [35].

4.5 CONFIGURATION SECURITY

In most of this book, we view the programmability of an FPGA as an inherent advantage that provides a circuit implementation platform or a multi-purpose acceleration engine. However, this flexibility also increases the potential for intellectual property theft compared to custom ASIC hardware. SRAM-based FPGAs (the focus of this chapter), have volatile configuration memory; to retain configuration data, a battery must provide a constant power supply to the configuration bits. This configuration data is stored in memory (RAM or a PROM) external to the FPGA, and is loaded into the FPGA at power-up. Someone monitoring the wires between these structures could capture the configuration data flowing from memory to the reconfigurable device. They could then duplicate the circuit simply by loading that data onto a new chip. Design firms that create FPGA-based hardware want to protect their work (which may have required significant design time) and prevent reverse-engineering of their designs.

To discourage their unauthorized copying, FPGA configurations can be watermarked with a special signature based on the circuit designer and the purchasing customer [28]. Of course, the design can still be copied and reverse-engineered, but the watermark can help identify the source of the unauthorized copies.

Design security can also be provided by encrypting configuration data to obscure the employed design techniques and/or functionality [26]. Many FPGA vendors now support configuration encryption with special on-chip decryption hardware. The Xilinx Virtex-II, for example, uses triple-key DES [54], and Altera's Stratix-II [3], Actel's ProASIC3 [2], and Lattice's ECP2 [29] all support 128-bit AES configuration encryption. In all cases, the keys are stored in the FPGA, and encrypted configurations may only be loaded if they were encrypted with the same key as that stored in the device. For a Virtex-II device, a battery must be attached to the proper pins to retain the key when the device is not powered. In contrast, the Stratix-II, ECP2, and ProASIC3 devices use nonvolatile memory for key storage, eliminating the need for a separate battery.

For systems that do not require runtime reconfiguration, the opportunity to copy a design can be reduced in end-products by not transmitting the configuration data on probeable wires. Antifuse and Flash FPGAs, based on nonvolatile configuration memory structures, inherently retain configuration data on-chip once configured, avoiding the need to transfer the information for systems not using runtime reconfiguration.

4.6 SUMMARY

The difficulty of clock speed increases and power consumption concerns motivate reconfigurable computing as an important technique to advance digital design, implementing compute-intensive application tasks in reconfigurable hardware. However, the performance and power penalty of reconfiguration has the real potential to overwhelm its benefits. This chapter discussed a variety of methods proposed and used to reduce and in some cases remove reconfiguration overhead, including various configuration architecture designs, scheduling and caching techniques, and ways to reduce the configuration data size.

In many cases, several approaches can be combined to further reduce the overhead. For example, relocation and defragmentation architectural features facilitate advanced configuration scheduling mechanisms that load configurations in advance of their use to minimize processor stall time during reconfiguration. Likewise, a configuration cache can be combined with a relocation- and defragmentation-enabled design that uses a staging area, providing a wide path to configuration memory to decrease transfer time. This in turn can be combined with wildcarding to allow multiple identical rows or columns to be configured simultaneously. Such combined methods allow reconfigurable computing system designers to effectively minimize reconfiguration overhead and to provide the full benefit of reconfigurable computing in future computing systems.

References

[1] A. Abnous, H. Zhang, M. Wan, G. Varghese, V. Prabhu, J. Rabaey. The Pleiades architecture. *Application of Programmable DSPs in Mobile Communications*, A. Gatherer, A. Auslander, eds., Wiley, 2002.

[2] Actel Corp. *ProASIC3 Flash Family FPGAs*. Actel Corp., Mountain View, CA, 2006.

[3] Altera, Inc. *Stratix-II*$^{\text{TM}}$ *Device Handbook, Volumes 1 and 2*, Altera, Inc., San Jose, 2005.

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

[5] Atmel Corp. *AT40K Series FPGA Interactive Architecture Guide*. Atmel Corp., San Jose, 1999.

[6] K. Bazargan, R. Kastner, M. Sarrafzadeh. Fast template placement for reconfigurable computing systems. *IEEE Design and Test, Special Issue on Reconfigurable Computing* 17(1), 2000.

[7] G. Brebner, O. Diessel. Chip-based reconfigurable task management. *International Conference on Field Programmable Logic and Applications*, 2001.

[8] J. Burns, A. Donlin, J. Hogg, S. Singh, M. de Wit. A dynamic reconfiguration runtime system. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[9] J. M. P. Cardoso, M. Weinhardt. From C programs to the Configure-Execute model. *Design, Automation, and Test in Europe*, 2003.

[10] E. Caspi, A. DeHon, J. Wawrzynek. A streaming multithreaded model. *Third Workshop on Media and Stream Processors*, 2001.

[11] D. Chang, M. Marek-Sadowska. Partitioning sequential circuits on dynamically reconfigurable FPGAs. *IEEE Transactions on Computers* 48(6), 1999.

[12] M. C.-T. Chao, G.-M. Wu, I.-H.-R. Jiang, Y.-W. Chang. A clustering- and probability-based approach for time-multiplexed FPGA partitioning. *IEEE/ACM International Conference on Computer-Aided Design*, 1999.

[13] M. M. Chu. *Dynamic Runtime Scheduler Support for SCORE*, Master's thesis, University of California, Berkeley, 2000.

[14] K. Compton, Z. Li, J. Cooley, S. Knol, S. Hauck. Configuration relocation and defragmentation for runtime reconfigurable systems. *IEEE Transactions on VLSI* 10(3), June 2002.

[15] A. Dandalis, V. K. Prasanna. Configuration compression for FPGA-based embedded systems. *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2001.

[16] M. Dales. Managing a reconfigurable processor in a general purpose workstation environment. *Conference on Design, Automation, and Test in Europe*, 2003.

[17] A. DeHon. DPGA utilization and application. *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 1996.

[18] O. Diessel, H. E. Gindy, M. Middendorf, H. Schmeck, B. Schmidt. Dynamic scheduling of tasks on partially reconfigurable FPGAs. *IEE Proceedings—Computers and Digital Techniques, Special Issue on Reconfigurable Systems* 147(3), 2000.

[19] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. R. Taylor. PipeRench: A reconfigurable architecture and compiler. *IEEE Computer* 33(4), April 2000.

[20] J. D. Hadley, B. L. Hutchings. Design methodologies for partially reconfigured systems. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[21] M. Handa, R. Vemuri. An efficient algorithm for finding empty space for online FPGA placement. *Design Automation Conference*, 2004.

[22] S. Hauck, Z. Li, E. J. Schwabe. Configuration compression for the Xilinx XC6200 FPGA. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.

[23] S. Hauck. Configuration prefetch for single context reconfigurable coprocessors. *ACM/SIGDA International Symposium on Field-Programmable Gate Arra*ys, 1998.

[24] J. R. Hauser. *Augmenting a Microprocessor with Reconfigurable Hardware*, Ph.D. thesis, University of California, Berkeley, 2000.

[25] Z. Huang, S. Malik. Managing dynamic reconfiguration overhead in systems-on-a-chip design using reconfigurable datapaths and optimized interconnection networks. *Design, Automation, and Test in Europe*, 2001.

[26] T. Kean. Cryptographic rights management of FPGA intellectual property cores. *International Symposium on Field-Programmable Gate Arrays*, 2002.

[27] A. Khan, N. Miyamoto, T. Ohkawa, A. Jamak, S. Kita, K. Kotani, T. Ohmi. An approach to realize time-sharing of flip-flops in time-multiplexed FPGAs. *IEEE International Conference on Field-Programmable Technology*, 2004.

[28] J. Lach, W. H. Mangione-Smith, M. Potkonjak. Fingerprinting techniques for field-programmable gate array intellectual property protection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20(10), 2001.

[29] Lattice Semiconductor Corp. *LatticeECP2 Family Data Sheet*, Lattice Semiconductor Corp., Hillsboro, OR, 2006.

[30] Z. Li, S. Hauck. Don't care discovery for FPGA configuration compression. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 1999.

[31] Z. Li, K. Compton, S. Hauck. Configuration caching for FPGAs. *IEEE Symposium on FPGAs for Custom Computing Machines*, 2000.

[32] Z. Li, S. Hauck. Configuration compression for Virtex FPGAs. *IEEE Symposium on FPGAs for Custom Computing Machines*, 2001.

[33] Z. Li, S. Hauck. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2002.

[34] R. Lysecky, F. Valid. A configurable logic architecture for dynamic hardware/software partitioning. *Design, Automation, and Test in Europe*, 2004.

[35] U. Malik, O. Diessel. On the placement and granularity of FPGA configurations. *IEEE International Conference on Field-Programmable Technology*, 2004.

[36] W. H. Mangione-Smith. ATR from UCLA. Personal communication, 1999.

[37] 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.

[38] J. Noguera, R. M. Badia. HW/SW codesign techniques for dynamically reconfigurable architectures. *IEEE Transactions on VLSI Systems* 10(4), 2002.

[39] J. Noguera, R. M. Badia. Multitasking on reconfigurable architectures: Microarchitecture support and dynamic scheduling. *ACM Transactions on Embedded Computing Systems* 3(2), May 2004.

[40] V. Nollet, P. Coene, D. Verkest, S. Vernalde, R. Lauwereins. Designing an operating system for a heterogeneous reconfigurable SoC. *Reconfigurable Architecture Workshop*, 2003.

[41] H. Quinn, L. S. King, M. Leeser, W. Meleis. Runtime assignment of reconfigurable hardware components for image processing pipelines. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003.

[42] 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.

[43] C. R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. M. Arnold, M. Gokhale. The NAPA adaptive processing architecture. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.

[44] H. Simmler, L. Levinson, R. Männer. Multitasking on FPGA coprocessors. *International Conference on Field-Programmable Logic and Applications*, 2000.

[45] H. Singh, G. Lu, M.-H. Lee, F. Kurdahi, N. Bagherzadeh, E. Filho, R. Mastre. MorphoSys: Case study of a reconfigurable computing system targeting multimedia applications. *Design Automation Conference*, 2000.

[46] 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.

[47] S. Trimberger, D. Carberry, A. Johnson, J. Wong. A time-multiplexed FPGA. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[48] H. Walder, M. Platzner. Non-preemptive multitasking on FPGAs: Task placement and footprint transform. *International Conference on Engineering of Reconfigurable Systems and Architectures*, 2002.

[49] G. Wigley, D. Kearney. The first real operating system for reconfigurable computing. *Australasian Computer Systems Architecture Conference*, 2001.

[50] M. J. Wirthlin, B. L. Hutchings. A dynamic instruction set computer. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[51] M. J. Wirthlin, B. L. Hutchings. Sequencing run-time reconfigured hardware with software. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 1996.

[52] G.-M. Wu, J.-M. Lin, Y.-W. Chang. Generic ILP-based approaches for time-multiplexed FPGA partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20(10), 2001.

[53] Xilinx, Inc. *XC6200 Field Programmable Gate Arrays Product Descriptio*n, Xilinx, Inc., San Jose, 1997.

[54] Xilinx, Inc. *Virtex-II Platform FPGAs: Complete Data Sheet*, Xilinx, Inc., San Jose, 2004.

[55] Xilinx, Inc. *Virtex-5 FPGA Configuration User Guide*, Xilinx, Inc., San Jose, 2006.

[56] G. Brebner. The swappable logic unit: A paradigm for virtual hardware, *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[57] W. Fu, K. Compton. An execution environment for reconfigurable computing. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005.

[58] M. Wirthlin, B. Hutchings. Improving functional density through run-time constant propagation. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 86–92, 1997.