CHAPTER 2

RECONFIGURABLE COMPUTING ARCHITECTURES

Lesley Shannon
School of Engineering Science
Simon Fraser University

There has been considerable research into possible reconfigurable computing architectures. Alternatives range from systems constructed using standard off-the-shelf field-programmable gate arrays (FPGAs) to systems constructed using custom-designed chips. Standard FPGAs benefit from the economies of scale; however, custom chips promise a higher speed and density for custom-computing tasks. This chapter explores different design choices made for reconfigurable computing architectures and how these choices affect both operation and performance. Questions we will discuss include:

- Should the reconfigurable fabric be instantiated as a separate coprocessor or integrated as a functional unit (see Instruction augmentation subsection of Section 5.2.2)
- What is the appropriate granularity (Chapter 36) for the reconfigurable fabric?

Computing applications generally consist of both control flow and dataflow. General-purpose processors have been designed with a control plane and a data plane to support these two requirements. All reconfigurable computers have a reconfigurable fabric component that is used to implement at least a portion of the dataflow component of an application.

In this discussion, the reconfigurable fabric in its entirety will be referred to as the *reconfigurable processing fabric*, or RPF. The RPF may be statically or dynamically reconfigurable, where *a static* RPF is only configured between application runs and a *dynamic* RPF may be updated during an application's execution.

In general, the reconfigurable fabric is relatively symmetrical and can be broken down into similar tiles or cells that have the same functionality. These blocks will be referred to as *processing elements*, or PEs. Ideally, the RPF is used to implement computationally intensive kernels in an application that will achieve significant performance improvement from the pipelining and parallelism available in the RPF. The kernels are called *virtual instruction configurations*, or VICs, and we will discuss possible RPF architectures for implementing them in the following section.

2.1 RECONFIGURABLE PROCESSING FABRIC ARCHITECTURES

One of the defining characteristics of a reconfigurable computing architecture is the type of reconfigurable fabric used in the RPF. Different systems have quite different granularities. They range from fine-grained fabrics that manipulate data at the bit level similarly to commercial FPGA fabrics, to coarse-grained fabrics that manipulate groups of bits via complex functional units such as ALUs (arithmetic logic units) and multipliers. The remainder of this section will provide examples of these architectures, highlighting their advantages and disadvantages.

2.1.1 Fine-grained

Fine-grained architectures offer the benefit of allowing designers to implement bit manipulation tasks without wasting reconfigurable resources. However, for large and complex calculations, numerous fine-grained PEs are required to implement a basic computation. This results in much slower clock rates than are possible if the calculations could be mapped to fewer, coarse-grained PEs. Fine-grained architectures may also limit the number of VICs that can be concurrently stored in the RPF because of capacity limits.

Garp's nonsymmetrical RPF

The BRASS Research Group designed the Garp reconfigurable processor as an MIPS processor and on-chip cache combined with an RPF [14]. The RPF is composed of an array of PEs, as shown in Figure 2.1. Unlike most RPF architectures, not all of the PEs (drawn as rounded squares in the array) are the same. There is one control PE in each row (illustrated as the dark gray square in the leftmost column) that provides communication between the RPF and external resources. For example, the control block can be used to generate an interrupt for the main processor or to initiate memory transactions. The remaining PEs (illustrated as light gray squares) in the array are used for data processing and modeled after the configurable logic blocks (CLBs) in the Xilinx 4000 series [13]. The number of columns of PEs is fixed at 24, with the middle 16 PEs dedicated to providing memory access for the RPF. The 3 extra PEs on the left and the 4 extra PEs on the right in Figure 2.1 are used for operations such as overflow, error checking, status checking, and wider data sizes.

The number of rows in the RPF is not fixed by the architecture, but is typically at least 32 [13]. A wire network is provided between rows and columns, but the only way to switch wires is through logic blocks, as there are no connections from one wire to another. Each PE operates at the bit level on two bits of data, performing the same operation on both bits based on the assumption that a large fraction of most configurations will be used for multibit operations. By creating identical configurations for both bits, the configuration size and time can be reduced but only at the expense of flexibility [13].

The loading of configurations into an RPF with a fine-grained fabric is extremely costly relative to coarse-grained architectures. For example, each PE

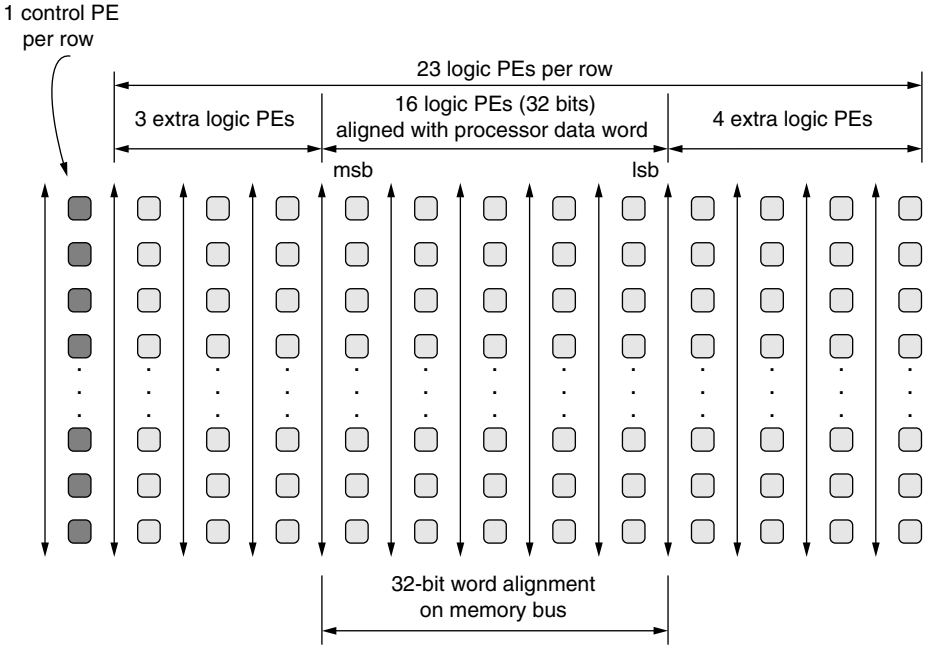

FIGURE 2.1 ■ Garp's RPF architecture. (*Source:* Adapted from [13].)

in Garp's RPF requires 64 configuration bits (8 bytes) to specify the sources of inputs, the PE's function, and any wires to be driven by the PE [13]. So, if there are only 32 rows in the RPF, 6144 bytes are required to load the configuration. While this may not seem significant given that the configuration bitstream of a commercial FPGA is on the order of megabytes (MB), it is considerable relative to a traditional CPU's context switch. For example, if the bit path to external memory from the Garp is assumed to be 128 bits, loading the full configuration takes 384 sequential memory accesses.

Garp's RPF architecture supports partial array configuration and is dynamically reconfigurable during application execution (i.e., a *dynamic* RPF). Garp's RPF architecture allows only one VIC to be stored on the RPF at a time. However, up to four different full RPF VIC configurations can be stored in the on-chip cache [13]. The VICs can then be swapped in and out of the RPF as they are needed for the application.

The loading and execution of configurations on the reconfigurable array is always under the control of a program running on the main (MIPS) processor. When the main processor initiates a computation on the RPF, an iteration counter in the RPF is set to a predetermined value. The configuration executes until the iteration counter reaches zero, at which point the RPF stalls. The MIPS-II instruction set has been extended to provide the necessary support to the RPF [13].

Originally, the user was required to write configurations in a textual language that is similar to an assembler. The user had to explicitly assign data and operations to rows and columns. This source code was fed through a program called the *configurator* to generate a representation for the configuration as a collection of bits in a text file. The rest of the user's source code could then be written in C, where the configuration was referenced using a character array initializer. This required some further assembly language programming to invoke the Garp instructions that interfaced with the reconfigurable array. Since then, considerable compiler work has been done on this architecture, and the user is now able to program the entire application in a high-level language (HLL) [14] (see Chapter 7).

2.1.2 Coarse-grained

For the purpose of this discussion, we describe coarse-grained architectures as those that use a bus interconnect and PEs that perform more than just bitwise operations, such as ALUs and multipliers. Examples include PipeRench and RaPiD (which is discussed later in this chapter).

PipeRench

The PipeRench RPF architecture [6], as shown in Figure 2.2, is an ALU-based system with a specialized reconfiguration strategy (Chapter 4). It is used as a coprocessor to a host microprocessor for most applications, although applications such as PGP and JPEG can be run on PipeRench in their entirety [8]. The architecture was designed in response to concerns that standard FPGAs do not provide reasonable forward compatibility, compilation time, or sufficient hardware to implement large kernels in a scalable and portable manner [6].

The PipeRench RPF uses pipelined configuration, first described by Goldstein et al. [6], where the reconfigurable fabric is divided into physical pipeline stages that can be reconfigured individually. Thus, the resulting RPF architecture is both partially and dynamically reconfigurable. PipeRench's compiler is able to compile the static design into a set of "virtual" stages such that each virtual stage can be mapped to any physical pipeline stage in the RPF. The complete set of virtual stages can then be mapped onto the actual number of physical stages available in the pipeline. Figure 2.3 illustrates how the virtual pipeline stages of an application can be mapped onto a PipeRench architecture with three physical pipeline stages.

A pipeline stage can be loaded during each cycle, but all cyclic dependencies must fit within a single stage. This limits the types of computations the array can support, because many computations contain cycles with multiple operations. Furthermore, since configuration of a pipeline stage can occur concurrent to execution of another pipeline stage, there is no performance degradation due to reconfiguration.

A row of PEs is used to create a physical stage of the pipeline, also called a physical stripe, as shown in Figure 2.2. The configuration word, or VIC, used to configure a physical stripe is also known as a virtual stripe. Before a physical

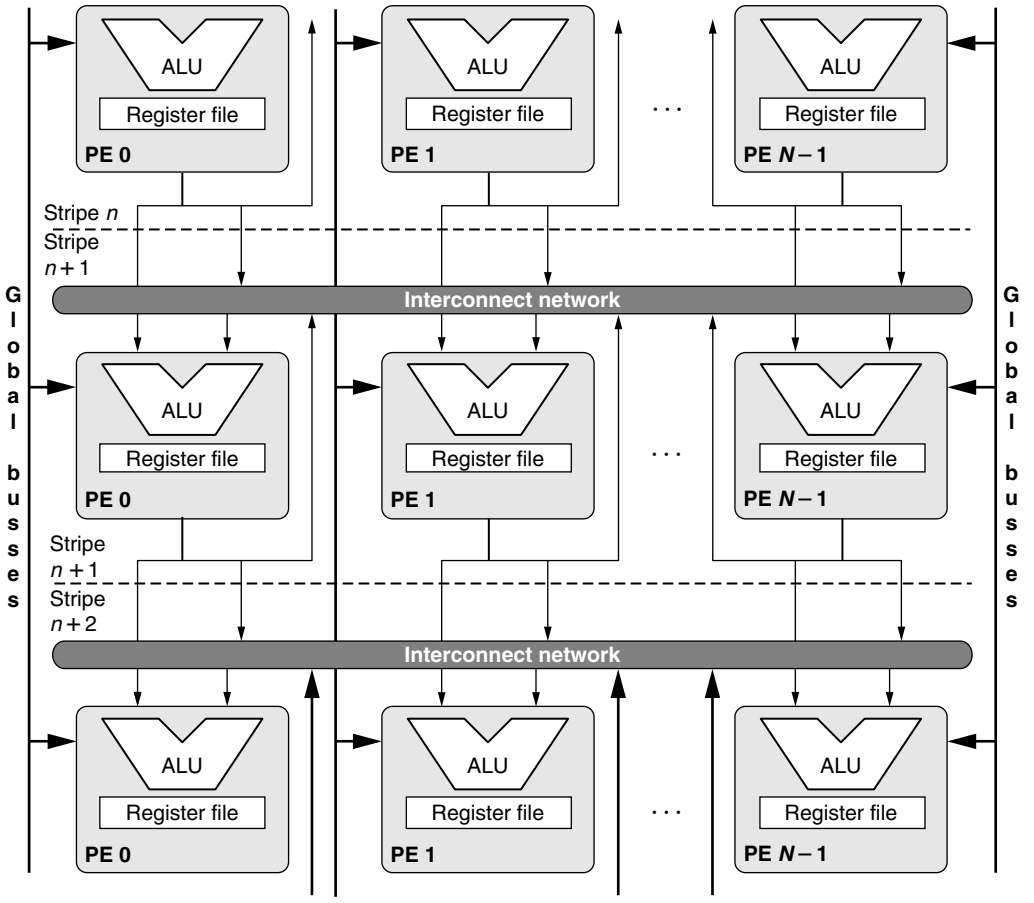

FIGURE 2.2 ■ PipeRench architecture: PEs and interconnect. (*Source:* Adapted from [6].)

stripe is configured with a new virtual stripe, the state of the present virtual stripe, if any, must be stored outside the fabric so it can be restored when the virtual stripe is returned to the fabric. The physical stripes are all identical so that any virtual stripe can be placed onto any physical stripe in the pipeline. The interconnect between adjacent stripes is a full crossbar, which enables the output of any PE in one stage to be used as the input of any PE in the adjacent stage [6].

The PEs for PipeRench are composed of an ALU and a pass register file. The pass register file is required as there can be no unregistered data transmitted over the interconnect network between stripes, creating pipelined interstripe connections. One register in the pass register file is specifically dedicated to intrastripe feedback. An 8-bit PE granularity was chosen to optimize the performance of a suite of kernels [6].

It has been suggested that reconfigurable fabric is well suited to stream-based functions (see Chapter 5, Section 5.1.2) and custom instructions [6]. Although

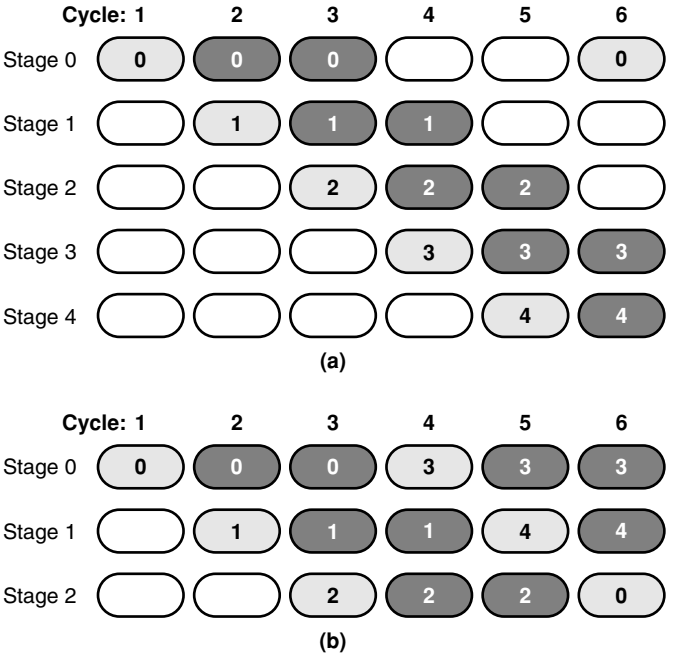

FIGURE 2.3 ■ The virtual pipeline stages of an application (a). The light gray blocks represent the configuration of a pipeline stage; the dark gray blocks represent its execution. The mapping of virtual pipeline stages to three physical pipeline stages (b). The physical pipeline stages are labeled each cycle with the virtual pipeline stage being executed. (*Source:* Adapted from [6].)

the first version of PipeRench was implemented as an attached processor, the next was designed as a coprocessor so that it would be more tightly coupled with the host processor [6]. However, the developers of PipeRench argue against making the RPF a functional unit on the host processor. They state that this could "restrict the applicability of the reconfigurable unit by disallowing state to be stored in the fabric and in some cases by disallowing direct access to memory, essentially eliminating their usefulness for stream-based processing" [6].

PipeRench uses a set of CAD tools to synthesize a stripe based on the parameters N, B, and P, where N is the number of PEs in the stripe, B is the width in bits of each PE, and P is the number of registers in a PE's pass register file. By adjusting these parameters, PipeRench's creators were able to choose a set of values that provides the best performance according to a set of benchmarks [6]. Their CAD tools are able to achieve an acceptable placement of the stripes on the architecture, but fail to achieve a reasonable interconnect routing, which has to be optimized by hand.

The user also has to describe the kernels to be executed on the PipeRench architecture using the *Dataflow Intermediate Language* (DIL), a single-assignment C-like language created for the architecture. DIL is intended for use by programmers and as an intermediate language for any high-level language compiler

that targets PipeRench architectures [6]. Obviously, applications have to be recompiled, and probably even redesigned, to run on PipeRench.

2.2 RPF INTEGRATION INTO TRADITIONAL COMPUTING SYSTEMS

Whereas the RPF in a reconfigurable computing device dictates the programmable logic resources, a full reconfigurable computing system typically also has a microprocessor, memory, and possibly other structures. One defining characteristic of reconfigurable computing chips is the integration, or lack of integration, of the RPF with a host CPU.

As shown in Figure 2.4, there are multiple ways to integrate an RPF into a computing system's memory hierarchy. The different memory components of the system are drawn as shaded rectangles, where the darker shading indicates a tighter coupling of the memory component to the processor. The types of RPF integration for these computing systems are illustrated as rounded

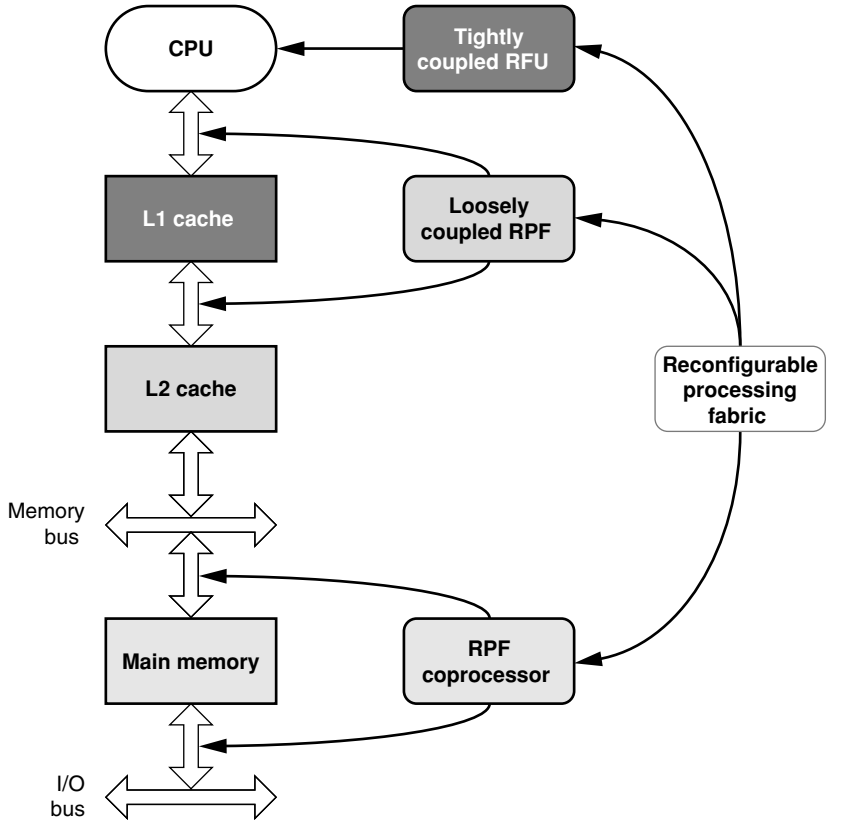

FIGURE 2.4 ■ Possible locations for the RPF in the memory hierarchy. (*Source:* Adapted from [6].)

rectangles, where the darker shading indicates a tighter coupling of the RPF to the processor. Some systems have the RPF as a separate processor [2–7]; however, most applications require a microprocessor somewhere to handle complex control. In fact, some separate reconfigurable computing platforms are actually defined to include a host processor that interfaces with the RPF [1]. Unfortunately, when the RPF is integrated into the computing system as an independent coprocessor, the limited bandwidth between CPU and reconfigurable logic can be a significant performance bottleneck.

Other systems include an RPF as an extra functional unit coupled with a more traditional processor core on one chip [8–24]. How tightly the RPF is coupled with the processor's control plane varies.

2.2.1 Independent Reconfigurable Coprocessor Architectures

Figure 2.5 illustrates a reconfigurable computing architecture with an independent RPF [1–7]. In these systems, the RPF has no direct data transfer links to the processor. Instead, all data communication takes place through main memory. The host processor, or a separate configuration controller, loads a configuration into the RPF and places operands for the VIC into the main memory. The RPF can then perform the computation and return the results back to main memory.

Since independent coprocessor RPFs are separate from the traditional processor, the integration of the RPF into existing computer systems is simplified. Unfortunately, this also limits the bandwidth and increases the latency of transmissions between the RPF and traditional processing systems. For this reason, independent coprocessor RPFs are well suited only to applications where the RPF can act independently from the processor. Examples include data-streaming applications with significant digital signal processing, such as multimedia applications like image compression and decompression, and encryption.

RaPiD

One example of an RPF coprocessor is the *Reconfigurable Pipelined Datapaths* [4], or *RaPiD*, class of architectures. RaPiD's RPF can be used as an independent

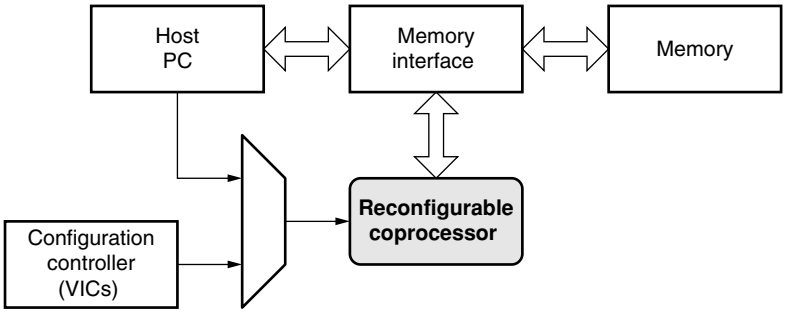

FIGURE 2.5 ■ A reconfigurable computing system with an independent reconfigurable coprocessor.

coprocessor or integrated with a traditional computing system as shown in Figure 2.5. RaPiD is designed for applications that have very repetitive pipelined computations that are typically represented as nested loops [5]. The underlying architecture is comparable to a super-scalar processor with numerous PEs and instruction generation decoupled from external memory but with no cache, no centralized register file, and no crossbar interconnect, as shown in Figure 2.6.

Memory access is controlled by the *stream generator*, which uses first-in-first-out (FIFOs), or *streams* (Chapter 5, Sections 5.1.2 and 5.2.1), to obtain and transfer data from external memory via the memory interface, as shown in Figure 2.7. Each stream has an associated address generator, and the individual address patterns are generated statically at compile time [5]. The actual reads and writes

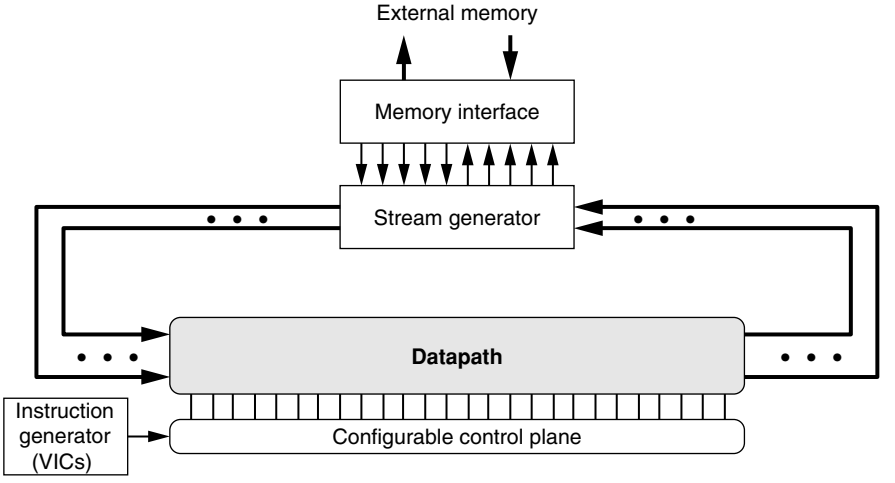

FIGURE 2.6 ■ A block diagram of the RaPiD architecture (*Source:* Adapted from [5].)

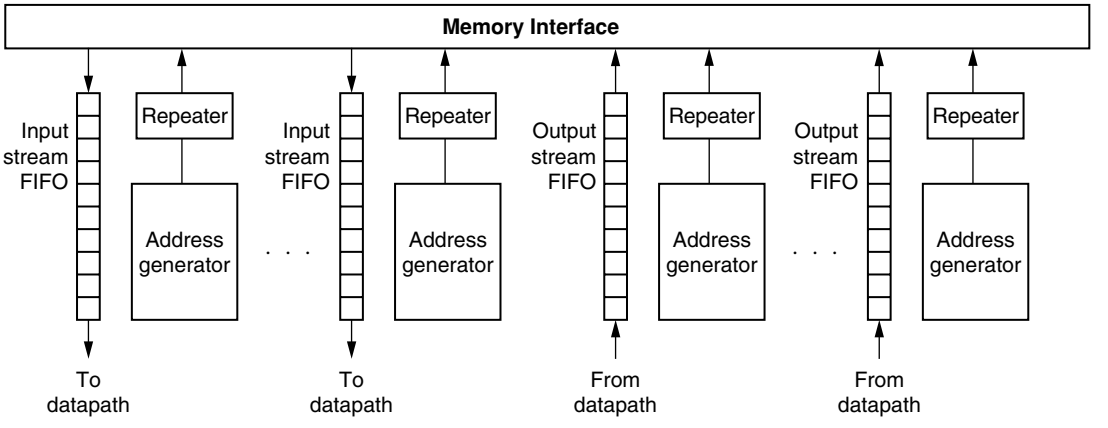

FIGURE 2.7 ■ RaPiD's stream generator. (*Source:* Adapted from [5].)

from the FIFOs are triggered by instruction bits at runtime. If the datapath's required input data is not available (i.e., the input FIFO is empty) or if the output data cannot be stored (i.e., the output FIFO is full), then the datapath will stall. Fast access to memory is therefore important to limit the number of stalls that occur. Using a fast static RAM (SRAM), combined with techniques, such as interleaving and out-of-order memory accesses, reduces the probability of having to stall the datapath [5].

The actual architecture of RaPiD's datapath is determined at fabrication time and is dictated by the class of applications that will be using the RaPiD RPF. This is done by varying the PE structure and the data width, and by choosing between fixed-point or floating-point data for numerical operations. The ability to change the PE's structure is fundamental to RaPiD architectures, with the complexity of the PE ranging from a simple general-purpose register to a multi-output booth-encoded multiplier with a configurable shifter [5].

The RaPiD datapath consists of numerous PEs, as shown in Figure 2.8. The creators of RaPiD chose to benchmark an architecture with a rather complex PE consisting of ALUs, RAMs, general-purpose registers, and a multiplier to provide reasonable performance [5]. The coarse-grained architecture was chosen because it theoretically allows simpler programming and better density [5]. Furthermore, the datapath can be dynamically reconfigured (i.e., a dynamic RPF) during the application's execution.

Instead of using a crossbar interconnect, the PEs are connected by a more area-efficient linear-segmented bus structure and bus connectors, as shown in Figure 2.8. The linear bus structure significantly reduces the control overhead—from the 95 to 98 percent required by FPGAs to 67 percent [5]. Since

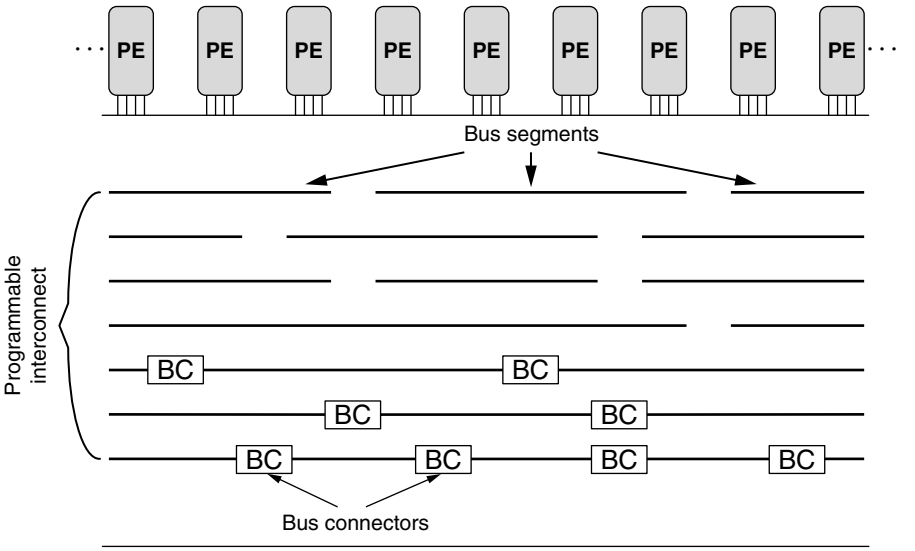

FIGURE 2.8 ■ An overview of RaPiD's datapath. (*Source:* Adapted from [5].)

the processor performance was benchmarked for a rather complex PE, the datapath was composed of only 16 PEs [5].

Each operation performed in the datapath is determined by a set of control bits, and the outputs are a data word plus status bits. These status bits enable data-dependent control. There are both *hard* control bits and *soft* control bits. As the hard control bits are for static configuration and are field programmable via SRAM bits, they are time consuming to set. They are normally initialized at the beginning of an application and include the tristate drivers and the programmable routing bus connectors, which can also be programmed to include pipelined delays for the datapath. The soft control bits can be dynamically configured because they are generated efficiently and affect multiplexers and ALU operations. Approximately 25 percent of the control bits are soft [5].

The instruction generator generates soft control bits in the form of VICs for the configurable control plane, as shown in Figure 2.9. The RaPiD system is built around the assumption that there is regularity in the computations. In other words, most of its processing time is spent within nested loops, as opposed to initialization, boundary processing, or completion [5], so the soft control bits are generated by a small programmable controller as a short instruction word (i.e., a VIC).

The programmable controller is optimized to execute nested loop structures. For each nested loop, the user's original code is statically compiled to remove all conditionals on loop variables and expanded to generate static instructions for loops [5]. The innermost loop can then often be packed into a single VIC with a count indicating how many times the VIC should be issued. One VIC can also be used to control more than one operation in more than one pipeline stage [5]. Figure 2.10(a) shows a snippet of code that includes conditional statements (if and for). This same functionality is shown in terms of static instructions in Figure 2.10(b).

As there are often parallel loop nests in applications, the instruction generator has multiple programmable controllers running in parallel (see Figure 2.9) [5]. Although this causes synchronization concerns, the appropriate status bits exist to provide the necessary handshaking. The VICs from each controller are

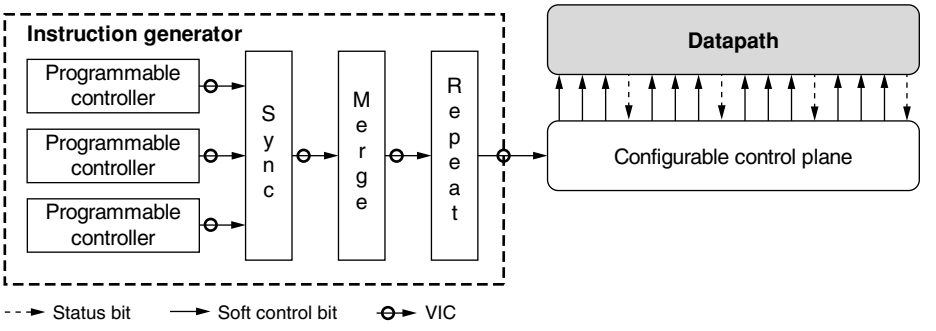

FIGURE 2.9 ■ RaPiD's instruction generator. (*Source:* Adapted from [5].)

```
for (i=0; i<10; i++)
{
      for(j=0; j<16; j++)
      {
              if(j==0)
                      load data;
              else if(j < 8)
                      x = x + y;
              else
                      z = y * z;
      }
}
```

(a)

```
Execute 10 times
{

     Execute once:          // j==0 case
            load data;
     Execute six times:     // 0<j<8 case
            x = x + y;
     Execute eight times:    // 7<j<16 case
            z = y * z;

}
```

(b)

FIGURE 2.10 ■ Original code (a) and pseudo-code (b) for static instruction implementation of the original code.

synchronized to ensure proper coordination between the parallel loops and then merged to generate the configurable control plane for the entire datapath [5].

There are obvious benefits to RaPiD, but it is not easily programmed: The programmer must use a specialized language and compiler designed specifically for RaPiD. This allows the designer to specify the application in such a way as to obtain better hardware utilization [5]. However, this class of architecture is not well suited to highly irregular computations with complex addressing patterns, little reuse of data, or an absence of fine-grained parallelism, which do not map well to RaPiD's datapath [5].

It is interesting to note that while RaPiD was implemented as a stand-alone processor, its creators suggest that it would be better to combine RaPiD with an RISC engine on the same chip so that it would have a larger application space [5]. The RISC processor could control the overall computation flow, and RaPiD could speed up the compute-intensive kernels found in the application. The developers also suggest that better performance could be achieved if RaPiD were a special functional unit as opposed to a coprocessor, because it would be more closely bound to the general-purpose processor [5]. These are the types of architecture we will be discussing in the following section.

2.2.2 Processor + RPF Architectures

As opposed to the independent coprocessor model, other systems more tightly couple the RPF with the host processor on the same chip; in some cases, the RPF is loosely coupled with the processor as an independent functional unit. Such architectures typically allow direct access to the RPF from the processor as well as independent access to memory, as do the Garp architecture [13] and the Chameleon system [20] (to be discussed in the following section). Alternatively, we can couple the RPF more tightly with the processor. For example, in architectures, such as Chimaera [18] (to be discussed later in this chapter), the

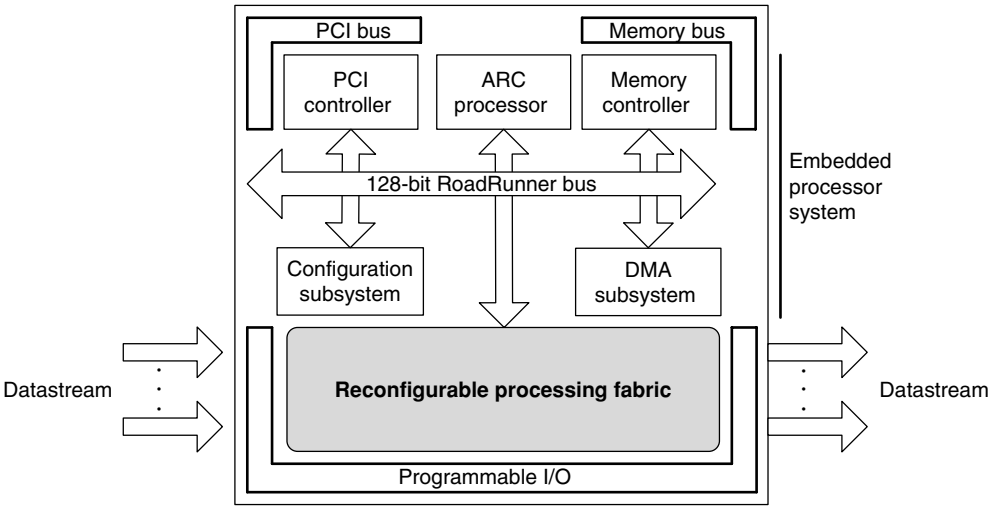

FIGURE 2.11 ■ Chameleon's RCP architecture. (*Source:* Adapted from a figure obtained off of Chameleon System's home page, which is no longer available.)

RPF is incorporated as a reconfigurable functional unit (RFU) (see Instruction augmentation subsection in Section 5.2.2) within the processor itself.

Loosely coupled RPF and processor architecture

The commercial Reconfigurable Communications Processor (RCP) was created by Chameleon Systems Inc. [20]. It combined an embedded processor subsystem with an RPF via a proprietary shared bus architecture, known as the RoadRunner bus (Figure 2.11). The RPF had direct access to the processor as well as direct memory access (DMA). The reconfigurable fabric also had a programmable I/O interface so that users could process off-chip I/O independent of the rest of the embedded on-chip processing system. This provided more flexibility for the RPF than in typical reconfigurable computing architectures, where the RPF generally had access only to the processor and memory.

The Chameleon architecture was able to provide improved price/performance relative to the highest-performing DSPs of its time, but its RCP consumed more power because of the RPF. After 2002, there was little mention of Chameleon or its RCP. Conceptually, the product was an interesting idea, but it failed to corner a product niche during the electronics market downturn.

Tightly coupled RPF and processor

Figure 2.12 illustrates a traditional processor's datapath architecture with the RPF integrated as an RFU. Such systems tightly couple the RFU to the central processing unit's (CPU) datapath similarly to the technology of traditional CPU functional units (FUs), such as the ALU, the multiplier, and the FPU. In some cases, these architectures only provide RFU access to input data from the register file in the same way as the traditional CPU FUs (Chimaera [18], PRISC

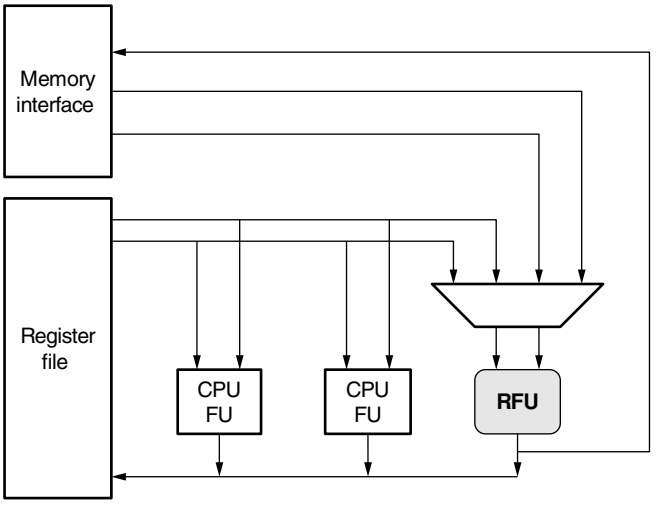

FIGURE 2.12 ■ The datapath of the processor + RFU architecture.

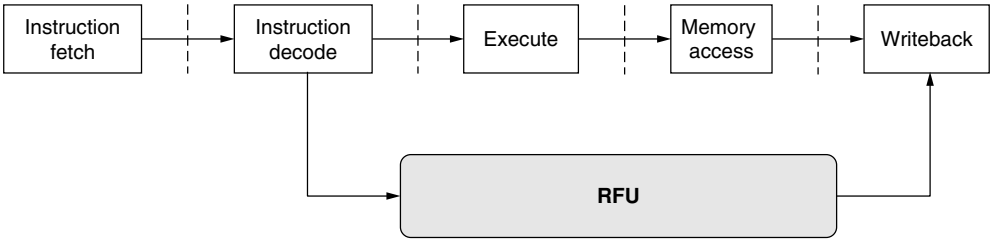

FIGURE 2.13 ■ An example of a pipeline of a processor with an RFU. (*Source:* Adapted from [16].)

[11], etc.). Other architectures allow the RFU to access data stored in the local cache/memory directly (e.g., OneChip [16]). Many of them can have multiple VICs instantiated in the RFU at once, enabling designers to accelerate multiple software instructions at the same time.

For reconfigurable computing architectures in which the RFU is tightly coupled with the processing core, the processor pipeline must be updated as shown in Figure 2.13. VICs in the RFU typically run during the *execute* stage (and possibly the *memory* stage) of the pipeline. Some of these processors are capable of running VICs in parallel with instructions that use more traditional processor resources, such as the ALU or FPU, and even support out-of-order execution (OneChip [16], Chimaera [18]).

Chimaera

The Chimaera architecture [18], shown in Figure 2.14, was developed at Northwestern University. Its developers created a C compiler that could create

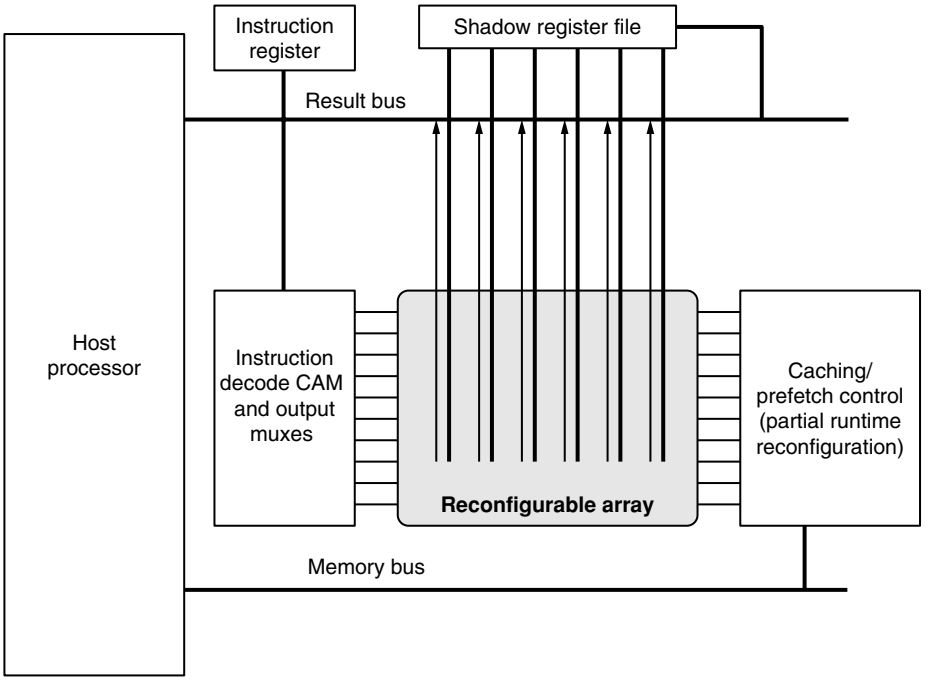

FIGURE 2.14 ■ Overview of the Chimaera architecture. (*Source:* Adapted from [18].)

specialized instructions for their RFU, known as RFUOPs (VICs for the purpose of our discussion) [19]. These custom instructions are created on a per application basis and have direct access to the processor's register file. Furthermore, commonly used VICs can be cached for easy reloading so that the processor does not have to stall while the RFU is configured [19].

The RFU is structured as a reconfigurable array (RA) of PEs, where any VIC occupies an integer number of rows. Influenced by the Triptych FPGA [18], the Altera Flex 8000 series, and the PRISC architecture [11], the array structure is FPGA-like to support computationally intensive kernels. Each PE in a row operates on 1 bit, with each row containing the same number of PEs as the size of the processor's memory word. The RFU can be partially configured so that multiple VICs can be cached in it at any given time. When an instruction is to be written to the RFU and there are no empty rows, the VIC that is overwritten is chosen such that configurations of the RFU will be minimized [19].

Another benefit of the Chimaera architecture is that it allows for speculative execution of VICs. Any VIC that is loaded in the RFU speculatively executes each cycle. If one of them is actually executed, the resulting value is stored at the writeback stage; otherwise, it is ignored and discarded. The RFU also supports multi-input operations, so that any VIC occupying one row will execute in a single clock cycle and with the appropriate data dependencies. Assuming that

data dependencies are not an issue, multi-cycle operations can execute without pipelining stalls [19].

When a VIC is detected at the decode stage of the pipeline, a check is made of the RFU to determine if it is already loaded. If it is not loaded, a check is made of the VIC cache. If the VIC instruction is not in either of these locations, it must be loaded from memory to reconfigure the necessary rows of the RFU. In that case the microprocessor will stall. This is time consuming because, although the precise configuration timing requirements are not specified, the objective is to minimize the number of configurations of the RFU performed from memory [19].

Chimaera has the benefit of a high-level design language for the user. It also has the same style interface as that of a normal stand-alone processor, which means that the architecture is able to provide extra functionality to improve performance, without complicating the design process. The idea is to treat the RFU as a cache for instructions as opposed to logic and then to assume that the majority of the functionality required for the algorithm will be supplied by the microprocessor [18]. In this way, the RFU can be used to accelerate the program's computationally intensive kernels. Integrating the RPF as an RFU within the processor has increased the bandwidth for communication between the two [18]. However, because the RFU cannot access memory directly, it is overly dependent on the host processor to fetch and store operands.

2.3 SUMMARY AND FUTURE WORK

In this chapter, we discussed key characteristics of reconfigurable computing architectures and their tradeoffs; specifically: (1) how the RPF should be coupled into the system, and (2) what the nature of the RPF should be. Fine-grained fabrics allow users to perform bitwise operations without wasting reconfigurable resources, whereas basic multibit computations can be mapped to fewer coarse-grained modules and run at a faster clock rate.

The coupling of the RPF with a traditional processor affects both its ability to do independent computation and the rate at which data can be transferred from the processor itself. Independent reconfigurable coprocessors are easily added to a traditional processing system and can operate independently from the processor. However, this loose coupling increases the latency and decreases the communication bandwidth between the processor and the RPF. In contrast, tightly coupling the RPF to the processor facilitates communication and data transfers, but limits the RFP's independence. In tightly coupled architectures, the RPF is often part of the processor's pipeline, potentially stalling execution until the VIC is completed. Loosely coupled RPFs try to offer the best of both worlds: sufficient independence from the main processor to prevent pipeline stalls combined with reasonable bandwidth for inter-processor/RPF communications.

One important challenge in developing reconfigurable computing architectures is to create CAD tools and programming environments that enable designers to use HLLS. This would allow designers to abstract the low-level hardware

of the RPF and to simplify programming the architecture, while still achieving speedup over a traditional processor. Another significant challenge is how to evaluate reconfigurable computing architectures. There is no equivalent to the Spec Benchmark [25] set for such evaluation. Furthermore, as these architectures may have different programming models or limited compiler support, designers are not easily able to run the same benchmark on multiple architectures for a standard comparison.

That Chameleon, and many other reconfigurable computing startup companies in similar market niches, was forced to close its doors during the electronic market downturn in the early 2000s illustrates an interesting aspect of reconfigurable computing as a whole. Even though, theoretically, special-purpose reconfigurable computing chips are a compelling technology, to date they have failed to achieve commercial success and there have been numerous failures. Many popular arguments have been used to justify this failure—they are too power-hungry; an effective high-level programming environment has not been developed; no one has identified a "killer" application to justify the design cost of using them—but no definitive answer exists. As it becomes increasingly difficult to improve the performance of traditional processor architectures, the possibility that reconfigurable computing architectures may yet find their place in the world of commercial success increases.

Despite the lack of significant market success to date, reconfigurable computing is still an area of significant ongoing research and commercial interest. For example, Rapport Inc.'s Kilocore design is a commercial derivative of the PipeRench architecture. As of 2007, Rapport was offering 256 PE components organized as 16 stripes, each composed of 16 8-bit PEs, and it has plans to expand its offerings to components containing thousands of PEs.

References

[1] J. M. Arnold. The Splash 2 software environment. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1993.

[2] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, S. Ghosh. PRISM-II compiler and architecture. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1993.

[3] M. J. Wirthlin, B. L. Hutchings. A dynamic instruction set computer. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1995.

[4] C. Ebeling, D. C. Cronquist, P. Franklin. RaPiD: Reconfigurable Pipelined Datapath. *Proceedings of the Sixth International Workshop on Field-Programmable Logic and Applications*, Springer-Verlag, September 1996.

[5] D. Cronquist, P. Franklin, C. Fisher, M. Figueroa, C. Ebeling. Architecture design of reconfigurable pipelined datapaths. *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, March 1999.

[6] S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, R. Laufer. PipeRench: A coprocessor for streaming multimedia acceleration. *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[7] H. Schmit. Incremental reconfiguration for pipelined applications. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1997.
[8] Y. Chou, P. Pillai, H. Schmit, J. Shen. PipeRench implementation of the instruction path coprocessor. *Proceedings of the 33rd International Symposium on Microarchitecture*, December 2000.
[9] M. J. Wirthlin, B. L. Hutchings, K. L. Gilson. The nano processor: A low resource reconfigurable processor. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1994.
[10] M. Budiu. Application-specific hardware: Computing without CPUs. *Fourth CMU Symposium on Computer Systems*, October 2001.
[11] R. Razdan, M. Smith. A high-performance microarchitecture with hardware-programmable functional units. *Proceedings of the 27th Annual IEEE/ACM International Symposium on Microarchitecture*, November 1994.
[12] B. Kastrup, A. Bink, J. Hoogerbrugge. ConCISe: A compiler-driven CPLD-based instruction set accelerator. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1999.
[13] J. Hauser, J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1997.
[14] T. J. Callahan, J. R. Hauser, J. Wawrzynek. The Garp architecture and C compiler. *Computer*, April 2000.
[15] R. D. Wittig, P. Chow. OneChip: An FPGA processor with reconfigurable logic. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, March 1996.
[16] J. E. Carrillo, E. P. Chow. The effect of reconfigurable units in superscalar processors. *Proceedings of the Ninth ACM International Symposium on Field-Programmable Gate Arrays*, February 2001.
[17] C. R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. A. Arnold, M. Gokhale. The NAPA adaptive processing architecture. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.
[18] S. Hauck, T. W. Fry, M. Hosier, J. P. Kao. The Chimaera reconfigurable functional unit. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1997.
[19] Z. A. Ye, A. Moshovos, S. Hauck, P. Banerjee. CHIMAERA: A high-performance architecture with a tightly coupled reconfigurable functional unit. *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.
[20] D. Wilson. Chameleon takes on FPGAs, ASICs. *Electronic Business Asia, EDN Online Magazine* (*http://www.edn.com/article/CA50551.html?partner=enews*), October 2000.
[21] P. Graham, B. Nelson. Reconfigurable processors for high-performance, embedded digital signal processing. *Proceedings of the Ninth International Workshop on Field-Programmable Logic and Applications*, August 1999.
[22] B. Salefski, L. Caglar. Reconfigurable computing in wireless. *Proceedings of the Design Automation Conference*, June 2001.
[23] T. Bijlsma, P. T. Wolkotte, G. J. M. Smit. An optimal architecture for a DDC. *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)—12th Reconfigurable Architecture Workshop (RAW 2006)*, April 2006.
[24] A. A. Chien, J. H. Byun. Safe and protected execution for the Morph/AMRM reconfigurable processor. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1999.
[25] Standard Performance Evaluation Corp. Spec Benchmarks (*http://www.spec.org*).