CHAPTER 9

STREAM COMPUTATIONS ORGANIZED FOR RECONFIGURABLE EXECUTION

André DeHon
Department of Electrical and Systems Engineering
University of Pennsylvania
Yury Markovskiy, Eylon Caspi, Michael Chu,
Randy Huang, Stylianos Perissakis, Laura Pozzi,
Joseph Yeh, John Wawrzynek
Department of Electrical Engineering and Computer Sciences
University of California–Berkeley

SCORE is a programming model for reconfigurable computing designed for application longevity and scalability, based on a streaming dataflow compute model (Section 5.1.3) and employing several system architectures (Section 5.2) to support scalability. The compute model allows us to abstract away hardware details such as platform capacity (e.g., number of lookup tables [LUTs]) and the detailed cycle-by-cycle timing of hardware implementation. This allows a single application description to automatically run faster on larger hardware or to fit onto smaller hardware. The abstraction of platform size and clock cycle timing makes SCORE a higher-level programming model than RTL-level descriptions such as VHDL (Chapter 6). The streaming dataflow model allows high concurrency and natural task descriptions for a large class of streaming applications, including signal and image processing.

Figure 9.1 shows one of the key scaling forms enabled. We capture the computation as a streaming dataflow graph of persistent operators (Section 5.1.2) abstracted from a particular platform (Figure 9.1(a)). On small hardware platforms, we use a phased reconfiguration manager (Phased reconfiguration manager subsection of Section 5.2.2) to implement the task as a sequence of configurations on the available hardware (Figure 9.1(b)). For larger platforms, more operators can be placed spatially, exploiting greater concurrency to reduce runtime (Figure 9.1(c)).

To achieve scalability, Stream Computations Organized for Reconfigurable Execution (SCORE) allows and encourages the programmer to ignore the hardware capacity of a particular platform and focus on capturing the fully spatial, streaming dataflow graph. A combination of the compiler and the runtime system must decompose and schedule the application onto a variety of hardware capacities. To support late-bound, runtime adaptation to various hardware platforms, the SCORE runtime employs a paged reconfiguration discipline (Section 9.2.4).

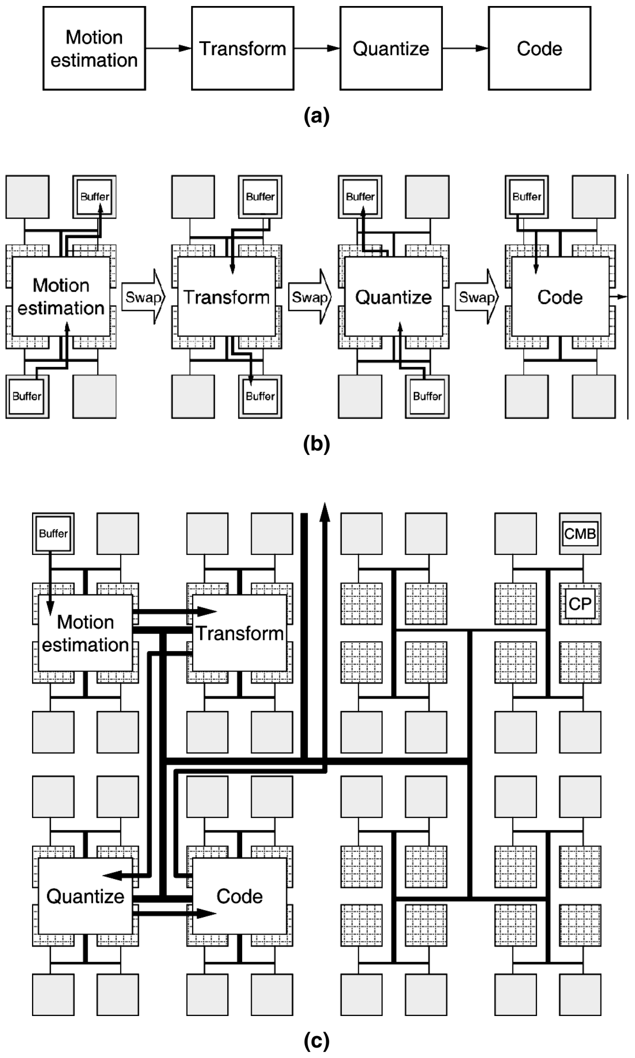


FIGURE 9.1 ■ Score application and sequential versus fully spatial execution: (a) a video compression task, (b) a capacity-limited sequential implementation, and (c) a fully spatial implementation on SCORE hardware.

In implementing this model, we must

- Provide concrete programming language instantiations for describing SCORE applications (Section 9.1).
- Select and employ suitable system architectures to implement the application and support area–time trade-offs for scalability (Section 9.2).
- Compile between the programming language description of the application and the runtime system architectures (Section 9.3).
- Provide runtime support for the tasks that must be performed during execution (Section 9.4).

The SCORE programming model demonstrates how compute model and system architectures come together to efficiently support a class of streaming applications.

9.1 PROGRAMMING

The specific compute model SCORE supports is Dynamic Streaming Dataflow with Allocation but without peeks (Dynamic streaming dataflow and Streaming dataflow with allocation subsections of Section 5.1.3), making it fully deterministic. Programs are composed by linking together operators (functions or objects, Section 5.1.2) and memory segments with first-in-first-out (FIFO) stream links (Section 5.1.3). Operators themselves can be described by their behavior or composed structurally as a graph.

Any number of languages that obey streaming dataflow semantics can be defined to program SCORE computations. The key requirements are to capture operators with appropriate dataflow input/output (I/O) interfaces and to allow operator compositions.

SCORE can be programmed with conventional programming languages (e.g., C++, Java) by defining stylized language subsets and library support to describe and compose SCORE operators. In Section 9.1.2, we show how to use C++ for dynamic composition.

In a multi-threaded language, such as Java or C++, with an appropriate thread package, a SCORE operator would be an independent thread that communicates with the rest of the program only through single-reader, single-writer I/O streams. Specifically, SCORE does not have a global, shared memory abstraction among operators (Single Memory Pool, Section 5.1.4). An operator may *own* a chunk of the address space (a memory segment) during operation and return it after it has completed, but no two operators may own a piece of memory simultaneously.

Alternately, SCORE programming could use a modern system-level design language, such as System C [1], as long as the communication library provides suitable dataflow communication semantics. To focus on the necessary semantics during SCORE development, we define an intermediate register transfer level (RTL) language to describe SCORE operators and their composition (Section 9.1.1). We view this intermediate language, TDF, as a device-independent, assembly language target on the way to platform-specific executable operators.

9.1.1 Task Description Format

Task Description Format (TDF) is basically an RTL-level operator description with special syntax for handling input and output datastreams from the operator [7, 22]. Common datapath operators can be described using a C-like syntax. For example, Figure 9.2 shows how an FIR computation might be implemented in TDF. Operators may have parameters whose values are bound at operator instantiation time; parameters are identified with the keyword `param`. In the

```
fir4(param signed[8] w0, param signed[8] w1,
     param signed[8] w2, param signed[8] w3,
     // param's bound at instantiation time
     input unsigned[8] x,
     output unsigned[20] y)
{
     state only(x): // "fire" when x present
     {
       // assignment to output y denotes a stream write
       y = w0*x + w1*x@1 + w2*x@2 + w3*x@3;
       // x@n notation picks out nth previous value for
       //   x on input stream.
       //  (this notation is patterned after Silage [2])
       goto only; // loop in this state
     }
}
```

FIGURE 9.2 ■ A TDF specification of 4-TAP FIR (a static rate operator).

FIR example, the coefficient weights are parameters; these are specified when the operator is created, and the values persist as long as the operator is used. The FIR reads from a single input stream (`x`) and produces a single output stream (`y`); the assignment to `y` denotes the stream write. The behavior of the state is gated on the arrival of the next `x` input value, producing a new `y` output for each such input.

To allow dynamic-rate dataflow (Dynamic streaming dataflow subsection of Section 5.1.3), the basic form of a behavioral TDF operator is that of a finite-state machine (FSM) (Finite State, Section 5.1.4), in which each state specifies the inputs that must be present before it can fire. Once the inputs arrive, the operator consumes them, and the FSM may choose to change states based on the input data consumed. A simple `merge` operator is shown in Figure 9.3 to demonstrate how the state machine can also be used to allow data-dependent consumption of input values. (Note: This version has been simplified for illustration; it does not properly handle the end-of-stream condition.) Output value production can be conditioned as illustrated in the `uniq` example shown in Figure 9.4. Together, data-dependent input consumption and output production allow the user to specify arbitrary, deterministic, dynamic-rate operators.

Of course, the FSM gives the user the semantic power to describe heavily sequential and complex, control-oriented operators. Nonetheless, the programmer should avoid sequentialization and complex control when possible, as operators with many states are less likely to use spatial computing resources efficiently. Larger operators can be composed structurally from smaller operators in a straightforward manner, as shown in Figure 9.5.

9.1.2 C++ Integration and Composition

With a suitable stream implementation and interface code, SCORE operators can be instantiated by and used with a conventional, multi-threaded programming

```
signed[w] merge(param unsigned[6] w,
               // can use parameters to define data width
                input signed[w] a,
                input signed[w] b)
{

  signed[w] tmpA; // define local state inside the operator
  signed[w] tmpB;
  // states used here to show dynamic data consumption
  state start(a,b): // requires inputs on both a and b to be
                    //   available in order to evaluate
   {
     // assignments to local variables have C-like semantics
     tmpA=a;  tmpB=b;
     if (tmpA<tmpB) { merge=tmpA;
                      goto replaceA; }
     else { merge=tmpB;
            goto replaceB; }
     // note: assignment to function name signifies a write to the
     //  output stream which is returned from operator instantiation
   }
  state replaceA(a): // requires availability of only input a
   {
     tmpA=a;
     if (tmpA<tmpB) { merge=tmpA;
                      goto replaceA; }
     else { merge=tmpB;goto replaceB; }
   }
 state replaceB(b): // requires availability of only input b
  {
    tmpB=b;
    if (tmpA<tmpB) { merge=tmpA;
                     goto replaceA; }
    else { merge=tmpB; goto replaceB; }
  }
}
```

FIGURE 9.3 ■ A TDF specification of `merge` operator (a dynamic input rate operator).

language. Figure 9.6 shows an example C++ program that uses the `merge` and `uniq` operators defined in Figures 9.3 and 9.4. Note that SCORE operator instantiation and composition can be performed in C++ code. Once created, the operators behave as independently running threads, operating in parallel with the main C++ execution thread. In general, a SCORE operator will run until its input streams are closed or its output streams are released (i.e., the stream is deallocated with a `free`-like operation).

After primitive behavioral (or leaf) operators have been defined (e.g., in TDF or some other suitable form) and compiled into their hardware-level implementation, large programs can be composed entirely in a conventional programming language as just described and illustrated in Figure 9.6. If one thinks of TDF as a portable assembly language for critical computational building blocks, then this language binding allows a high-level language to compose these building blocks

```
// uniq behaves like the unix command of the same name;
//  it filters an input stream, removing any adjacent, duplicate
//  entries before passing them on to the output stream.
signed[w] uniq(param unsigned[6] w,
               input signed[w] x)
{
   signed[w] lastx;
   state start(x):
     { lastx=x; uniq=x; goto loop;}
   state loop(x):
     {
       if (x=!lastx)
         { lastx=x; uniq=x; }
      goto loop;
     }
}
```

FIGURE 9.4 ■ A TDF specification of `uniq` operator (a dynamic output rate operator).

```
merge3uniq(param unsigned[6] n,
           input signed[n] a,
           input signed[n] b,
           input signed[n] c,
           output signed[n] o)
{
         signed [n] t;
         t=merge(n,merge(n,a,b),c);
         o=uniq(n,t);
}
```

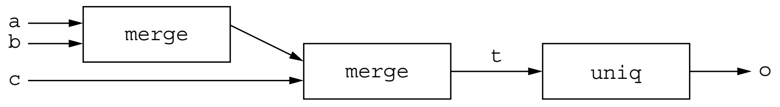
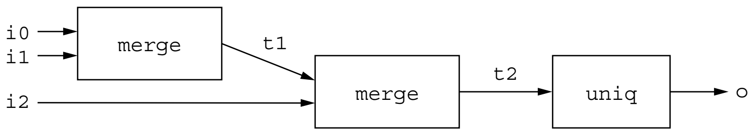


FIGURE 9.5 ■ The TDF compositional operator.

in much the same way that assembly language kernels are composed using high-level languages in order to efficiently program early DSPs and supercomputers. The instantiation parameters for TDF operators allow the definition of generic operators that can be highly customized to the needs of the application.

9.2 SYSTEM ARCHITECTURE AND EXECUTION PATTERNS

To support the SCORE programming model efficiently, implementations are based on several system architectures and execution design patterns (e.g., DeHon et al. [3]). In this section, we highlight how these architectures are used and introduce additional execution patterns.



```
#include "Score.h"
#include "merge.h"
#include "uniq.h"
int main()
{
  char data0[] = { 3, 5, 7, 7, 9 };
  char data1[] = { 2, 2, 6, 8, 10 };
  char data2[] = { 4, 7, 7, 10, 11 };
  // declare streams
  SIGNED_SCORE_STREAM i0,i1,i2,t1,t2,o;
  // create 8-bit wide input streams
  i0=NEW_SIGNED_SCORE_STREAM(8);
  i1=NEW_SIGNED_SCORE_STREAM(8);
  i2=NEW_SIGNED_SCORE_STREAM(8);
  // instantiate operators
  //  note: instantiation passes parameters and streams to the operators
  t1=merge(8,i0,i1);
  t2=merge(8,t1,i2);
  o=uniq(8,t2);
  // alternately, we could use: new merge3uniq (8,i0,i1,i2,o);
  // write data into streams
  //  (for demonstration purposes;
  //    real streams would be much longer and not come from main)
  for (int i = 0; i < 5; i++) {
    STREAM_WRITE(i0, data0[i]);
    STREAM_WRITE(i1, data1[i]);
    STREAM_WRITE(i2, data2[i]);
  }
  STREAM_CLOSE(i0); // close input streams
  STREAM_CLOSE(i1);
  STREAM_CLOSE(i2);
  // output results (for demonstration purposes only)
  for (int cnt=0; !STREAM_EOS(o); cnt++) {
    cout << "result["<< cnt << "]=" <<
       STREAM_READ(o) << endl;
 }
 STREAM_FREE(o);
 return(0);
}
```

FIGURE 9.6 ■ An example of instantiation and usage in C++.

9.2.1 Stream Support

SCORE heavily leverages the stream abstraction (Chapter 5, Section 5.1.3) for communication between operators. The streamed data can be assigned to a buffer if the producer and consumer are not coresident (see Figure 9.1(b));

if they are coresident, the data can be assigned to physical networking (see Figure 9.1(c)). Further, any number of mechanisms (e.g., shared bus, packet-switched network, time-multiplexed network, configured links) can implement the stream based on data rate, predictability, and platform capabilities. Once data communication is organized as a stream, the platform knows which data to prefetch and how to package it to or from memory.

When a SCORE implementation physically implements streams as wires between dynamic-rate operators, data presence (Data presence subsection of Section 5.2.1) tags allow us to abstract out dynamic data rates or delays. While data presence allows producers to signal consumers that data are not ready, it is often useful to signal the opposite direction as well; consequently, we also implement a *back-pressure* signal, which allows the consumer to inform the producer that it is not ready to consume additional inputs. We can further place queues between the producer and the consumer to decouple their cycle-by-cycle firing.

When the consumer is not ready, produced values accumulate in the queue, allowing the producer to continue operation; if there are stored values in the queue, the consumer can continue to operate while the producer is stalled as well. Queues are of finite size, so a full queue also uses back-pressure to stall an attached producer. In dynamic data rate operations where queue size cannot be bounded (Dynamic streaming dataflow subsection of Section 5.1.3), the hardware signals the OS when queues fill, and the OS may need to allocate additional queue capacity at runtime to prevent deadlock [4].

9.2.2 Phased Reconfiguration

When the operator graph is too large for the platform, it is necessary to share the physical hardware in time (see Figures 9.1(b) and 9.7). For a reconfigurable platform, this can be done by changing the configuration overtime, to implement the

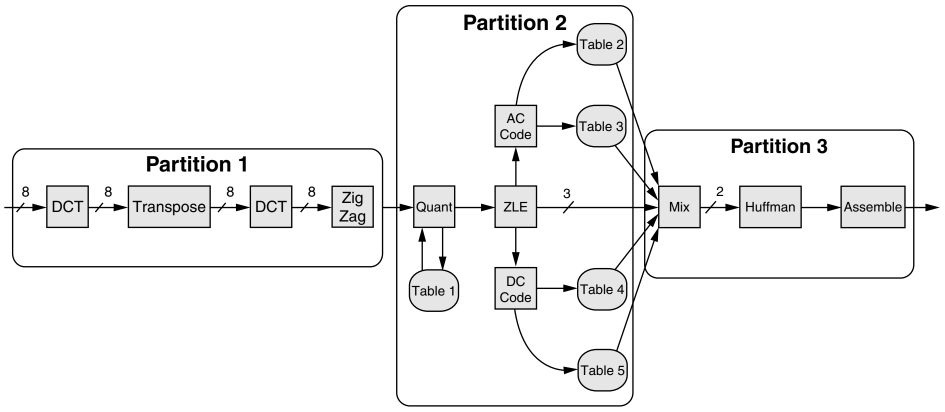


FIGURE 9.7 ■ Partitioning of a JPEG image encoder to match platform capacity.

graph in pieces (Phased reconfiguration manager subsection of Section 5.2.2). Reconfiguration, however, can be an expensive operation requiring many cycles. To minimize its overhead cost, we want to run each operator for many cycles between reconfigurations. In particular, if we can ensure that each operation runs for a large number of cycles compared to the reconfiguration time, then we can make the overhead for reconfiguration small ($T_{\texttt{run-before-reconfig}} >> T_{\texttt{config}}$). Streaming data with large queues helps us achieve this. We can queue up a large number of data items that will keep the operator busy. We then reconfigure the operator, compute on the queued data, and, if the consumer is not coresident, queue up the results (Figure 9.1(b)). When the input queue is empty or the output queue is full, we reconfigure to the next set of operators.

9.2.3 Sequential versus Parallel

When the platform contains both processors and reconfigurable logic, it is possible to assign some operators to the processor(s) (Processor subsection of Section 5.2.2) and some to the reconfigurable fabric. We can compile SCORE operators either to processor instructions or to reconfigurable configurations, and we can even save both implementations as part of the program executable. At load time or runtime, low-throughput operators can be assigned to the sequential processor(s), while high-throughput logic can be assigned to the reconfigurable fabric. As the size of the reconfigurable fabric grows, more operators can be implemented spatially on it.

Phased reconfiguration can be ineffective when mutually dependent cycles are large compared to the size of the platform. Processors are designed to time-multiplex their hardware at a fine granularity; thus, one way to fit large operator cycles onto the platform is to push lower throughput operators onto the processor until the cycle is contained.

We interface the processor to the reconfigurable array using a streaming coprocessor arrangement (Streaming Coprocessors, Section 5.2.1). The processor can write data into stream FIFOs to go to the reconfigurable array coprocessor, and it reads data back from them. This decouples the cycle-by-cycle operation of the reconfigurable array from the processor, abstracting the relative timing of the two units. In the case where the reconfigurable array can be occupied (e.g., allocated to another operator or task), this reduces coresidence requirements between operators on the array and processor. As a result, the options for the array size to vary among platform implementations increase.

9.2.4 Fixed-size and Standard I/O Page

To allow the platform size to vary with the implementation platform, it is necessary to perform placement at load time or runtime based on the amount of physical hardware and the time-multiplexed schedule. If we had to place everything at the LUT level, we would have a very large placement problem. Further, if we allowed partial reconfiguration in order to efficiently support the fact that different operators may need to be resident for different amounts of time, we would have a fragmentation and bin-packing problem [5], as different operators take up different space and have different footprints. We can simplify the runtime problem by using a discipline of fixed-size pages that have a standard I/O interface.

First, we decide on a particular page size (e.g., 512 4-LUTs) for the architecture. At compile time, we organize operators into standard page-size blocks so that we can perform the intrapage placement and routing offline at compile time. At runtime, we simply place pages and perform interpage routing. The runtime placement problem is simplified because all pages are identically sized and interchangeable. Furthermore, because pages are typically 100 to 1000 4-LUTs, the runtime placement problem is two to three orders of magnitude smaller than LUT-level placement. Unfortunately, fixed-size pages may incur internal fragmentation, leaving some resources in each page unused. Brebner's SLU is an early example of this pattern [6].

Note that this is the same basic approach used in virtual memory, where we do not manage every bit or even every word independently, but instead gather a fixed number of words into a page and manage (e.g., map and swap) them as a group. In both cases, this reduces the overhead associated with page mapping considerably.

9.3 COMPILATION

We have developed a complete compilation flow from TDF to conventional FPGAs using Verilog (an HDL similar to VHDL—see Chapter 6) as an intermediate form (Figure 9.8) [7]. The TDF compiler, `tdfc`, automatically generates RTL Verilog to efficiently implement the streaming constructs of the TDF language, including flow control checking, stream buffering in queues, and stream pipelining. The TDF compiler also maps between abstract operators of arbitrary size and the fixed-size pages supported at runtime by the system architecture.

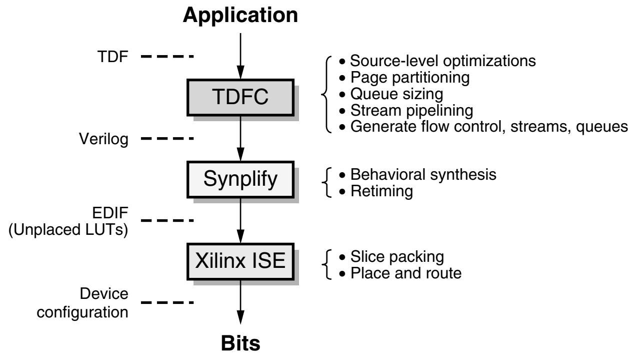


FIGURE 9.8 ■ TDF compilation flow targeting an FPGA.

The compiler then emits a netlist of pages for compilation by a commercial backend FPGA synthesis, place, and route flow.

Because SCORE streams abstract the number of clock cycles between operators, we can pipeline both the interconnect between operators and the operator datapaths. To pipeline operators, the compiler adds registers to the input and output streams and employs retiming (Chapter 18) to redistribute the registers into the operator logic.

To accommodate the wide range of operator sizes that the programmer may produce, the compiler must perform operator packing and splitting in order to target any particular, fixed-size page. Our previous experience suggests that most user-written leaf operators require fewer than 512 4-LUTs, which means that page packing will be adequate to reshape most applications. Many large operators are feedforward pipelines (e.g., DCT, IDCT), which can be easily decomposed using directional cuts in the dataflow. For the general case, it is necessary to decompose large state machines to fit them onto small pages. This could be done by starting with individual states and clustering state logic and datapaths, obeying the page area and I/O bound. To minimize delay, the goal is to group states that typically execute together so as to minimize the frequency of state transitions that cross the page boundary. Clustering techniques such as those described by Li et al. [8] can be employed for this general clustering case.

9.4 RUNTIME

To support the late-bound task and platform mapping integral to SCORE's power and scalability, we must perform scheduling, placement, and routing no earlier than load time. In this section, we highlight how these tasks can all be performed quickly at load time or runtime.

9.4.1 Scheduling

We support SCORE's virtualization model in the presence of late-bound platform mapping with a load-time and runtime scheduler. We do not know the capacity of the platform until load time; consequently, we cannot partition the graph into sets of pages that fit on the platform before then. Further, because operators have dynamic execution times and dynamic consumption and production rates, the relative execution time of each operator cannot be known with certainty until execution. To support SCORE efficiently, we must be able to:

- Quickly partition the page graph into platform-feasible components (within milliseconds).
- Produce a high-quality schedule—that is, one that minimizes the time to run the task (minimizes the make span).
- Minimize the sequential handling required for managing reconfiguration and advancing the schedule.

In the simplest cases, we partition the graph once, at load time, when the program starts and never again. In this way, we amortize the cost of partitioning across the entire application runtime (Figure 9.9). If the application will run for seconds, we can afford tens of milliseconds for this scheduling operation while keeping the overhead small. If we can decrease the scheduling time, then it will be possible to run even shorter jobs efficiently. In more advanced cases, the graph may change during execution, or the execution rates of operators may change in a data-dependent way. In such cases, it might be useful to repartition and reschedule the graph during execution. The shorter we can make the partitioning time, the more frequently we can afford to invoke the partitioner without paying a large overhead.

We have developed a series of schedulers to address these issues [9–12]. Our highest-quality scheduler (shown in Figure 9.10) is quasi-static and load-time based [10], and operates in two phases: (1) load-time partitioning and (2) runtime schedule advancement. At application load-time, the scheduler partitions the page-level dataflow graph into platform-feasible subgraphs. This partition can use feedback information on operator and stream activity rates based on previous program runs. The load-time partitioning heuristic requires only a few hundred thousand processor cycles (e.g., submillisecond time on gigahertz processors) for graphs with up to one hundred operators [12].

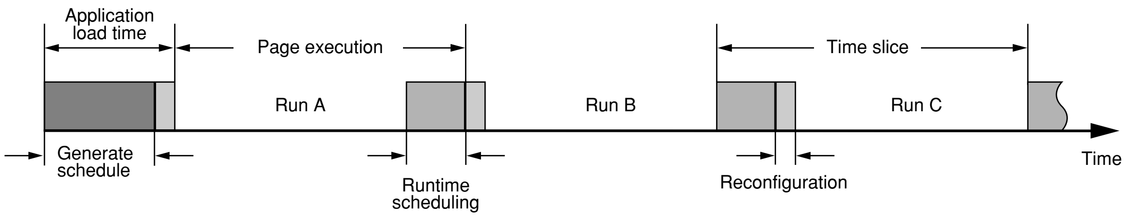


FIGURE 9.9 ■ An application execution timeline.

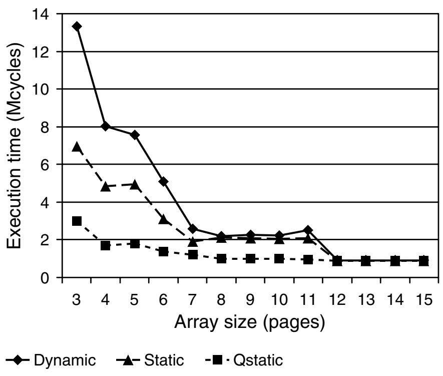


FIGURE 9.10 ■ JPEG decoder scaling: Total execution time is compared among fully dynamic, fully static, and quasi-static schedulers.

The result is a schedule for the phased reconfiguration. During execution, the runtime system advances the computed schedule by reconfiguring the array at regular intervals (Phased reconfiguration manager subsection of Section 5.2.2). The schedule computed at load time specifies a nominal period for each schedule timeslice. Additionally, the system monitors execution to determine when the current configuration can no longer make forward progress (e.g., all input buffers are empty or all output buffers are full) and dynamically triggers early phase termination and schedule advancement.

9.4.2 Placement

Using the fixed-size and standard I/O pages discipline (Section 9.2.4) we immediately reduce the size of the placement task by two to three orders of magnitude. Nonetheless, the placement task may still take too long when run using conventional single-processor-based placers at reconfiguration time or even load time. Fortunately, once we have a spatially parallel reconfigurable computing platform, we can use the platform itself to perform placement substantially faster. In Wrighton and DeHon [13] and Wrighton [14], we show how to perform simulated annealing spatially with reconfigurable logic; we can place a graph with 1000 movable elements in roughly 1 million cycles. Even if we only ran the placement engine at 100 MHz, this would mean that we could perform placement in 10 ms. If each page held 512 4-LUTs, this would correspond to platforms with half a million 4-LUTs.

The key idea for spatial simulated annealing is to build a placement engine on top of the reconfigurable platform. If we make each page large enough, then it can act as a cellular placement cell. As a placement cell, it holds a candidate, logical page and negotiates exchanges with its nearest neighbors (i.e., cellular automata system architecture Section 5.2.5). A pair of adjacent pages will swap logical pages if they estimate that the swap will produce a superior placement (e.g., shorter wire lengths) or if the randomness in the simulated annealing process suggests attempting the swap anyway. All pages can be paired up and can negotiate swaps in parallel, allowing many moves per swap epoch.

By pairing up only neighbors, we can guarantee minimizing the interconnect for this placement engine and keep the cycle times short. Because there is one cellular placement cell for every page site on the device, the hardware and parallelism in the placement engine scales exactly to the size of the placement problem that needs to be solved. Wrighton and DeHon [13] estimate that 400 4-LUTs are adequate to implement a 100 MHz cellular placement cell on Xilinx Virtex-II–generation hardware [15]; this suggests that SCORE platforms with 512 LUT pages will be able to perform their own placement.

9.4.3 Routing

Once the pages have been placed, we must perform interpage routing. Again, we can exploit the fact that we have a spatially parallel computing platform to route tasks in 100,000 to 1 million cycles [16]. Here, we augment the interpage network with additional logic to allow it to identify all free paths between a source node and a sink node in parallel. This permits a flooding search (e.g., Figure 9.11) to find a free path in the time it takes to propagate a signal across the network rather than the time it takes to perform a sequential search on a large graph structure in memory. Consequently, each new path can be added in tens of cycles rather than the tens of thousands of cycles required by the best software routers.

Using randomization, rip-up, and multiple restarts, this approach can even perform congestion negotiation and achieve comparable quality to PathFinder [17] (Chapter 17), the state-of-the-art software-routing algorithm for FPGAs [18, 19]. With word-wide (e.g., 16-bit) datapaths for the interpage network, the additional area overhead for this augmented network is less than 30 percent when network routing channels are switch-area limited; the augmented network adds only control wires, so it has almost no area overhead when network-routing channels are wire dominated.

An alternate approach is to employ a packet-switched network for interpage routing (see Marescaux et al. [20] and Kapre et al. [21]) to avoid the need to compute and configure the network. Packet switches are generally much larger and have higher latency than configured switches, but they may be able to handle multirate and dynamic traffic more efficiently.

Figure 9.11 shows the result of a path search for a route from node 4 to node 2. Light thick lines show preexisting routes; dark thick lines show the free paths explored between source and sink. At the crossover switchbox (labeled "XXX"), only a single switch is found by both source- and sink-initiated searches.

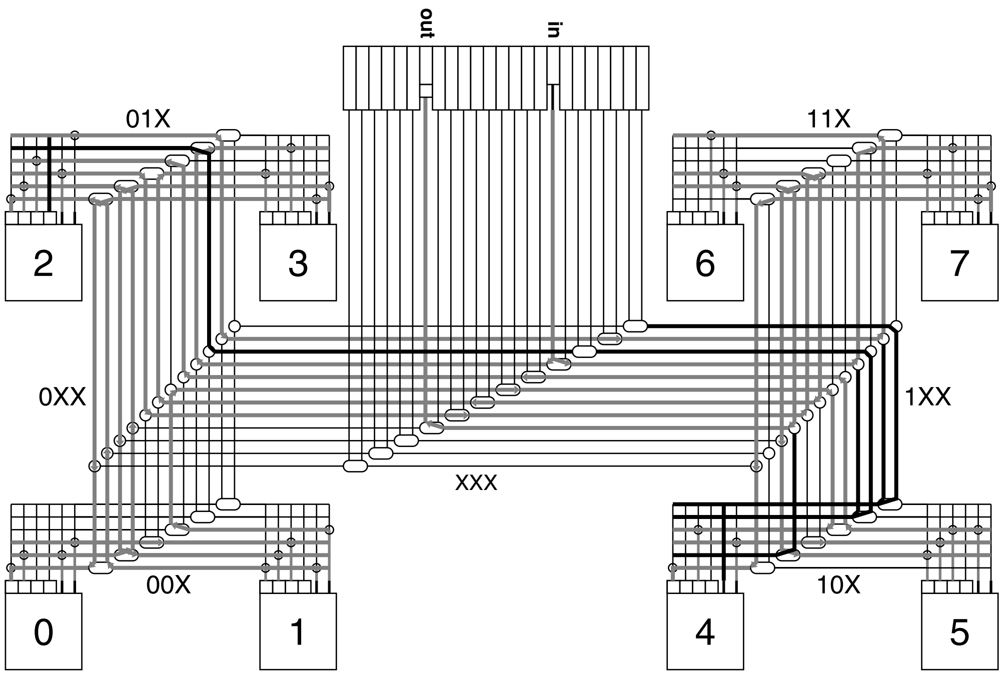


FIGURE 9.11 ■ A spatially parallel path search.

9.5 HIGHLIGHTS

SCORE compilation has automatically mapped image-processing applications (e.g., wavelet, JPEG, MPEG) to streamed implementations that exceed 100 MHz sample throughput on a Virtex-II Pro XC2VP70-7 [12]. In comparable technology, a 4-page SCORE design outperforms a Pentium-3 (500 MHz) by 10 times on JPEG compression. Mapped design performance scales to deliver larger speedup with additional pages (see Figure 9.10).

For further details on SCORE, see DeHon et al. [12] and Caspi et al. [22].

References

[1] Open System C Initiative. *System C 2.1 Language Reference Manual*, May 2005 (*http://www.systemc.org*).

[2] D. Genin, J. Rabaey, P. Hilfinger, C. Scheers, H. DeMan. DSP specification using the SILAGE language. *Proceedings of the IEEE ICASSP Conference*, April 1990.

[3] A. DeHon, J. Adams, M. deLorimier, N. Kapre, Y. Matsuda, H. Naeimi, M. Vanier, M. Wrighton. Design patterns for reconfigurable computing. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[4] T. M. Parks. *Bounded Scheduling of Process Networks*, UCB/ERL95–105, University of California, Berkeley, 1995.

[5] K. Bazargan, R. Kastner, M. Sarrafzadeh. Fast template placement for reconfigurable computing systems. *IEEE Design and Test of Computers* 17(1), January–March 2000.

[6] G. Brebner. The swapable logic unit: A paradigm for virtual hardware. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1997.

[7] E. Caspi. *Design Automation for Streaming Systems*, Ph.D. thesis, University of California, Berkeley, 2005.

[8] Z. Li, K. Compton, S. Hauck. Configuration caching techniques for FPGAs. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

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

[10] 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. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 2002.

[11] Y. Markovskiy. *Quasi-Static Scheduling for SCORE*, Master's thesis, University of California, Berkeley, December 2004.

[12] A. DeHon, Y. Markovskiy, E. Caspi, M. Chu, R. Huang, S. Perissakis, L. Pozzi, J. Yeh, J. Wawrzynek. Stream computations organized for reconfigurable execution. *Journal of Microprocessors and Microsystems* 30(6), September 2006.

[13] M. Wrighton, A. DeHon. Hardware-assisted simulated annealing with application for fast FPGA placement. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 2003.

[14] M. Wrighton. *A Spatial Approach to FPGA Cell Placement by Simulated Annealing*, Master's thesis, California Institute of Technology, June 2003.

[15] Xilinx, Inc. *Xilinx Virtex-II 1.5V Platform FPGAs Data Sheet*, San Jose, July 2002.

[16] A. DeHon, R. Huang, J. Wawrzynek. Hardware-assisted fast routing. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002.

[17] L. McMurchie, C. Ebeling. PathFinder: A negotiation-based performance-driven router for FPGAs. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 1995.

[18] R. Huang, J. Wawrzynek, A. DeHon. Stochastic, spatial routing for hypergraphs, trees, and meshes. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 2003.

[19] A. DeHon, R. Huang, J. Wawrzynek. Stochastic spatial routing for reconfigurable networks. *Journal of Microprocessors and Microsystems* 30(6), September 2006.

[20] T. Marescaux, V. Nollet, J.-Y. Mignolet, A. Bartic, W. Moffat, P. Avasare, P. Coene, D. Verkest, S. Vernalde, R. Lauwereins. Run-time support for heterogeneous multitasking on reconfigurable SOCs. *INTEGRATION, The VLSI Journal* 38(1), October 2004.

[21] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, A. DeHon. Packet-switched vs. time-multiplexed FPGA overlay networks. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2006.

[22] E. Caspi, M. Chu, R. Huang, N. Weaver, J. Yeh, J. Wawrzynek, A. DeHon. Stream Computations Organized for Reconfigurable Execution (SCORE): Introduction and tutorial (*http://www.cs.berkeley.edu/projects/brass/documents/score_tutorial.html*); a short version appears in *FPL '2000* (Lecture Notes in Computer Science, 1896), 2000.