Available online at www.sciencedirect.com
SCIENCE DIRECT®
Microprocessors and Microsystems 30 (2006) 334–354
MICROPROCESSORS AND MICROSYSTEMS
www.elsevier.com/locate/micpro

Stream computations organized for reconfigurable execution

André DeHon *, Yury Markovsky, Eylon Caspi, Michael Chu, Randy Huang, Stylianos Perissakis, Laura Pozzi, Joseph Yeh, John Wawrzynek

University of California at Berkeley, Berkeley, CA 94720-1776, USA

Available online 3 March 2006

Abstract

Reconfigurable systems can offer the high spatial parallelism and fine-grained, bit-level resource control traditionally associated with hardware implementations, along with the flexibility and adaptability characteristic of software. While reconfigurable systems create new opportunities for engineering and delivering high-performance programmable systems, the traditional approaches to programming and managing computations used for hardware systems (e.g., Verilog, VHDL) and software systems (e.g., C, Fortran, Java) are inappropriate and inadequate for exploiting reconfigurable platforms. To address this need, we develop a stream-oriented compute model, system architecture, and execution patterns which can capture and exploit the parallelism of spatial computations while simultaneously abstracting software applications from hardware details (e.g., timing, device capacity, and microarchitectural implementation details) and consequently allowing applications to scale to exploit newer, larger, and faster hardware platforms. Further, we describe hardware and software techniques that make this late-bound platform mapping viable and efficient.

Keywords: FPGA; Reconfigurable; Scalability; Design reuse; Streaming; System architecture; Design patterns; Pipe-and-filter; Productivity

1. Introduction

Reconfigurable computing offer a large potential advantage in computing power over conventional, microprocessor based systems [24]. Uniprocessor performance improvements have stalled (e.g., [1]) while silicon capacity continues to grow; consequently, the raw computational density gap widens and the role for spatially oriented programmable computations grows larger. Unfortunately, FPGA-based and reconfigurable computing remains limited to niche applications (e.g., [56,68,78]) and ASIC replacements for rapid prototyping, fast time-to-market, and low non-recurring engineering (NRE). The key factor limiting wider application of reconfigurable computing is the difficulty of developing high-performance, high-quality reconfigurable solutions and the manual effort required to scale reconfigurable solutions to newer, larger, and more capable hardware.

To unleash the power of reconfigurable computing and allow spatially programmable architectures to play a full-featured role alongside microprocessors, it is necessary to exploit the key advantages of the reconfigurable hardware while abstracting implementation details to facilitate scaling. We need a compute model that abstracts hardware details:

- device capacity
- resource placement
- inter- and intra-module timing
- microarchitecture of computing blocks

At the same time, the model and execution architecture should allow applications to

- ride vendor technology curves
- reuse components
- exploit parallelism, especially pipelined, spatial dataflow
- exploit bit-level control

* Corresponding author.
E-mail addresses: andre@acm.org (A. DeHon), yurym@cs.berkeley.edu (Y. Markovsky), eylon@cs.berkeley.edu (E. Caspi), mmchu@cs.berkeley.edu (M. Chu), rhuang@cs.berkeley.edu (R. Huang), sper@cs.berkeley.edu (S. Perissakis), laura.pozzi@unisi.ch (L. Pozzi), jyeh@cs.berkeley.edu (J. Yeh), johnw@cs.berkeley.edu (J. Wawrzynek).
URL: http://brass.cs.berkeley.edu/ (A. DeHon).

0141-9331/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2006.02.009

A compute model allows us to define the meaning of an application separately from a particular hardware device, just as the sequential instruction set architecture (ISA) model defines the meaning of an application separately from the detailed processor microarchitecture [3].

The ISA abstraction has served us well for four decades, allowing processors to scale in capacity while preserving software developments. However, its sequential, instruction-oriented execution and monolithic memory model is now limiting the further exploitation of silicon capacity. It does not capture the freedom for efficient, spatial execution that exists in many applications, and hence cannot be used to fully exploit the capabilities of FPGAs and reconfigurable computing systems.

Conventional register-transfer-level (RTL) hardware models, and their associated programming languages (e.g., VHDL, Verilog) are not the solution, either. Their execution model is cycle-based, focusing on exactly when each computation occurs; this hides freedom which exists in most applications to pipeline computations to accommodate changing hardware costs (e.g., increasing interconnect delays [67,66]). Furthermore, an RTL description captures a particular level of parallelism for an application, corresponding to a particular amount of hardware. This assumes that one will discard and rewrite the application to provide higher performance on larger-capacity devices. This model thus inhibits automated scalability to newer devices with greater capacity and different relative operation timings.

To meet the opportunities of spatially programmable hardware, we employ a compute model (Section 2.2) which is more parallel than the ISA model while being more abstract than cycle-by-cycle RTL. The model facilitates automated run-time reconfiguration (RTR), abstracted from the user and scalable across different sized FPGA platforms (Section 2.5). To make parallel computation manageable, and to guide designers towards efficient, spatially parallel and scalable solutions, we employ a pipe-and-filter style (streaming) system architecture (Section 2.3). We support this model and system architecture with efficient execution patterns that exploit the capabilities of FPGA and reconfigurable substrates and facilitate scaling (Section 4). Several of the execution patterns can be layered on top of existing FPGA architectures, but the system architecture suggests ways to design and build novel reconfigurable computing devices specifically optimized to support this compute model and system architecture (Section 5). Late-bound platform mapping requires that we perform important mapping tasks (i.e., scheduling, placement, and routing) at load time or run time. Consequently, we introduce compilation techniques (Section 7) and run-time support (Section 8) that allow us to perform this mapping orders of magnitudes faster than conventional approaches.

We started developing stream computations organized for reconfigurable execution (SCORE) in 1999 and first presented it in [17,16].

2. Model and system architecture

In our model for scalable reconfigurable designs, there are two things we need to capture:

1. *What is the meaning of a program?* We need to capture the meaning abstracted from any particular implementation so as to enable automated optimization and scaling. Optimization and scaling, in turn, exploit the freedom allowed by the compute model to provide efficient implementations.
2. *How should programmers think about a scalable, application for reconfigurable computing systems?* In addition to abstracting the user from the hardware, we also need to guide the designer towards good solutions:
   - solutions that are efficient on the platform (high performance, minimal cost)
   - solutions that are scalable
   - solutions and components that are reusable
   - solutions that minimize complexity and avoid pitfalls

   The *compute model* (Section 2.2) defines the meaning of the computation. The *system architecture* (Section 2.3) provides the disciplined framework for designing applications which exploit the hardware and avoid challenges to scaling, reuse, and complexity management.

2.1. Motivation

2.1.1. What do spatially programmable architectures do well?

These architectures excel at doing the *same* thing over-and-over again. Their strength is in being able to wire up a datapath that performs some regular, core computation with direct dataflow. Reconfigurable architectures have higher computational density than processors precisely because they allocate minimal area for instruction storage [24,23]. The reconfigurable datapath can be inexpensively pipelined and can be reused to do the same calculation on different data. In fact, most applications perform the same operations on large amounts of data—consider the ratio between dynamic instructions issues and static instructions in a program description. Spatially programmable architectures allow us to identify the common dataflow, build it once on a compact, programmable fabric, and reuse it repeatedly. In effect, we hoist the description of the computation dataflow and the computation required to construct it out of the inner-loop, construct it once, and then perform just that necessary computation on each data set or data item.

2.1.2. What are the reusable building blocks and how are they composed?

In sequential programing, we write functions to define a reusable unit of computation. We compose function calls together in sequences and expressions in order to build larger functionality hierarchically out of components. This allows us to conquer the complexity of a large task by

dividing it into components, and the components are often common elements which we can abstract and reuse in multiple problems.

To exploit what these spatially programmable architectures do well, we want to avoid demanding sequential composition whenever possible. However, we can naturally compose blocks spatially, with the output of one block feeding into the input of the next. The blocks are computational pipelines, perhaps with state, transforming input sequences into output sequences. The blocks might be individual multipliers and adders which can be composed into larger blocks like FIRs, FFTs, or DCTs. Larger blocks may be further composed in a hierarchy, much as function calls may be composed to form new, larger functions.

2.1.3. What happens when we scale and how do we prepare computations for scaling?

When we scale applications, we can place more spatial pipelines simultaneously on our platform, and we can afford to make each block larger, perhaps exploiting more internal parallelism. So, ideally, we want to capture the whole computation as a set of concurrent, operating spatial pipelines (e.g., Figs. 1(a) and 2). When we must run the task on platforms which are smaller than the ideal computation, we can decompose the large, concurrent graph and run pieces of it in sequence (e.g., Fig. 1(b)). When the platform is large enough, the task can be implemented spatially on the platform (e.g., Fig. 1(c)). That is. rather than starting with a maximally sequential design, as ISAs have traditionally done, we start with maximally spatially parallel designs.

In order to scale across technology and platforms, we must also deal with variations in timing:

1. as we sequentialize the concurrent graph, we run different subsets of the graph concurrently and consequently change the relative timing of the data processing operations
2. we can further perform area-time tradeoffs within a block, changing its timing
3. we may introduce micro-architectural optimizations in the block implementation which changes its timing (e.g., adding or optimizing a carry chain to make arithmetic faster)

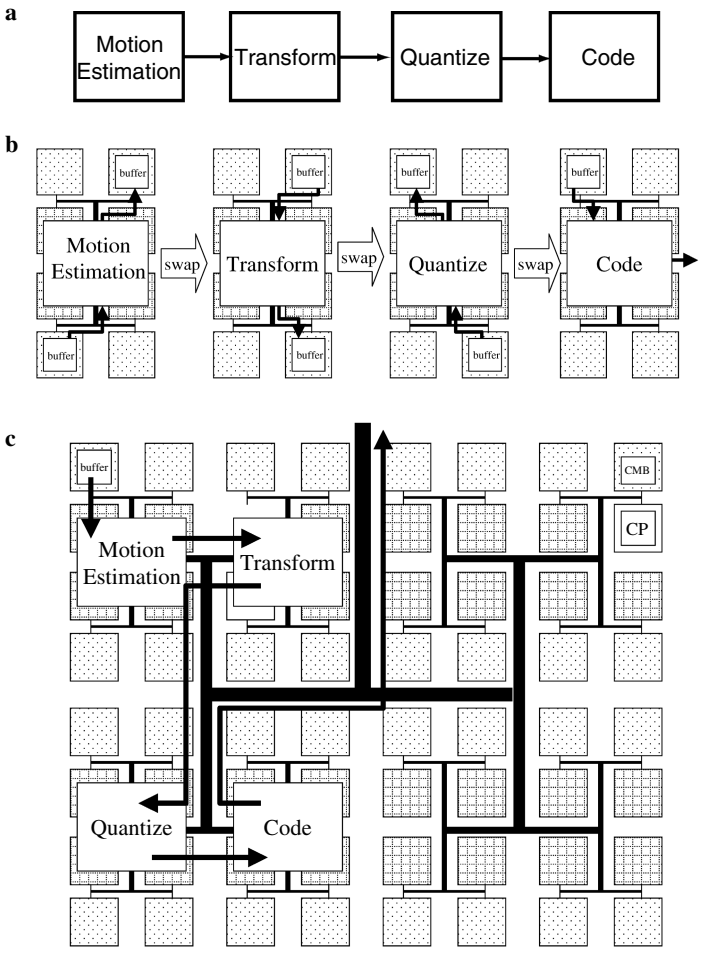

Fig. 1. SCORE application and sequential vs fully spatial execution. (a) Video compression task. (b) Capacity-limited, sequential implementation. (c) Fully spatial implementation on SCORE hardware.

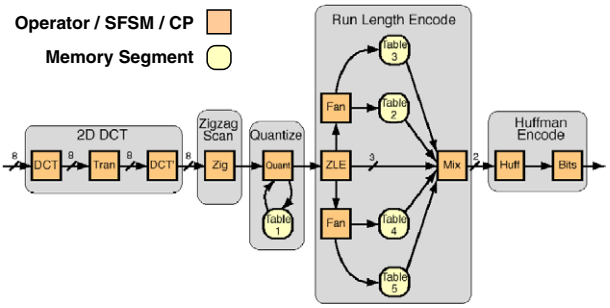

Fig. 2. JPEG image encoder.

4. as we scale to smaller feature sizes, computation scales faster than non-local communication, suggesting the need to change the relative timing allocation (e.g., number of clock cycles) between computation and communication

Consequently, we must tolerate timing variations in order to allow scaling. Note that ISA models tolerate variable memory delays as part of their strategy to facilitate scaling; for ISA-based processors, the memory delay and bus speeds scale at different rates compared to the computation.

2.2. Compute model

Motivated by these demands, we employ a stream-oriented compute model to facilitate the capture and scaling of compute-centric reconfigurable designs.

2.2.1. Informal description

A SCORE computation is a graph of computation nodes (*operators*) and memory blocks (*segments*) linked together by *streams* (e.g., Fig. 2). Streams provide node-to-node communication and are simply single-source, single-sink FIFO queues with unbounded length. Graph nodes (operators) are of two forms: (1) Finite-State Machine (SCORE FSM or *SFSM*) nodes which interact with the rest of the graph *only* through their stream links; and (2) Turing complete (SCORE TM or *STM*) nodes which support resource allocation in addition to stream operations.

SFSMs have the property that the present state identifies a set of inputs to be read from the input streams. Once a full set of inputs is present, the SFSM consumes the inputs from the appropriate set of input FIFOs and may conditionally emit outputs or close input or output streams. As with any standard FSM, SFSMs transition to a new state based on their inputs and present state. Each SFSM has a distinguished *done* state into which it may enter to signal its completion and to remove itself from the running computation. With its finite state, a SFSM is a natural abstraction for a spatially programmable logic on an FPGA or other reconfigurable platform.

An STM node is similar to a SFSM node but adds the ability to allocate memory segments and to create new graph nodes (SFSM or STM operators) and edges (streams) in the SCORE compute graph. With this additional capability, STMs may be best handled by a sequential processor in the reconfigurable systems. Allocation is an infrequent operation compared to datapath use, so it can tolerably be implemented on a sequential fabric without substantially impacting application performance. Here, we follow the system design principle of making the common cases fast and implementing the uncommon cases inexpensively.

Memory is allocated in finite-sized blocks called *segments*. Each segment may be owned by a single operator at a time. An STM may allocate new segments and pass them on to an SFSM or STM node that it creates. Upon termination, when a STM or SFSM node enters the *done* state, it returns ownership of any received segments back to the operator that created it. If an operator attempts to access a memory segment that it does not presently own, that access is blocked (i.e., the operator stalls) until the operator regains ownership of the memory segment.

A more formal treatment of the compute model is provided in [16,14].

2.2.2. Timing independence

The operational semantics of the SCORE compute model are fully deterministic. This follows from the determinism of individual operators, the timing independent communication discipline, and the fact that operators cannot side-effect each other's state. In particular, (1) operators communicate with each other only through streams whose token flow semantics guarantee a timing-independent order of execution; (2) memory segments have a single, unique owner at any time and thus do not suffer from multiple-user read/write-ordering hazards. Thus, the observable results of a SCORE computation are completely independent of the timing of any operator or the delay along any stream.

This timing independence is key to scalability, as it allows many things to vary from implementation to implementation without changing the behavior of the application. Implementations may vary in

- co-residence of operators – on smaller platforms, fewer operators are co-resident; on larger platforms, more are co-resident
- clock cycles between operators – as interconnect continues to scale more slowly than compute, the number of pipeline stages between physical operators will increase; this also abstracts placement, allowing operators to be placed varying distances apart
- implementation fabric – operators may be implemented on slower fabrics (e.g., sequentially on a processor) or faster fabrics (e.g., a specialized, hardwired unit for common operator types)

2.2.3. Relation to other models of computation

There are a large number of stream-oriented process network models with similar and slightly varying power

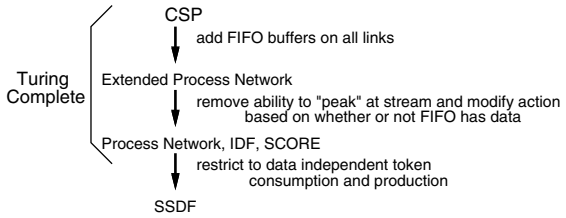

Fig. 3. Hierarchy of process network models.

and semantics [44,43]. Fig. 3 shows where SCORE is placed in power and semantics in relation to a number of common models. Hoare's communicating sequential processes (CSP) [36] is a very general model for capturing and modeling concurrent programs. CSP uses direct rendezvous between communicating processes, but one can add FIFO buffering by constructing explicit queue processes that mediate between the original computing processes. With buffering, the result is an extended process network model. Extended process network models allow operators to check if data is present and to act differently based on its presence. This ability, while powerful, makes it harder to verify that a program will work correctly on any hardware platform. That is, a program that works on one piece of hardware and one timing may produce a different result and hence may break when the relative timings of the hardware change. To avoid this, SCORE and numerous other process network models prohibit user-level operators from testing the presence of data on input streams. The resulting *blocking read* semantics force an operator to always wait for the desired data (a compiled operator may still peek at data for efficient implementation, provided it preserves the original semantics of the compute model). SCORE graphs have the same semantic power as process network models such as Integer Data Flow (IDF) [11] and Kahn process networks [40,41]. Process networks, including SCORE, allow dynamic data rates between operators; this makes them semantically more powerful than Statically Schedulable Data Flow, such as Synchronous Dataflow (SDF) [45,7] and Cyclo-Static Dataflow (CSDF) [8]. The expressive power of dynamic data rates enables efficient implementation of operations with data-dependent input and output sizes (e.g., packet filtering, compression, decompression), but it comes at the cost that tasks cannot be optimally scheduled offline and that no upper bound can be placed on the buffer depth between operators [59]. Process network models such as IDF and SCORE are Turing Complete while SSDF models are not.

2.3. System architecture

While we have explicitly detailed the computational model for SCORE above, SCORE can alternately be viewed as a *pipe-and-filter* system architecture restriction on a general multithreading or CSP compute model.

We could have chosen a general multithreading model or even CSP as our base compute model. However, in so doing, we would offer developers little guidance on how to write good applications that can be implemented efficiently on spatially programmable hardware and take advantage of the strengths of these platforms [43]. Further, we would have subjected the programmer to all the challenges and pitfalls that notoriously make multithreaded parallel programming hard, such as manual synchronization and subtle, lock-related bugs.

Instead, we introduced a stylization which matches the strengths of reconfigurable hardware. If a developer can match his problem to a set of persistent, stream-connected operators, there is a good chance it can run efficiently on spatially programmable hardware. Further, we give the developer a discipline for communication which provides strong guarantees for size and timing independence.

Parallel programming is hard because there is *too much* freedom for the developer to manage. Strategically restricting the programming model helps to constrain the developer to a portion of the design space where many good solutions can be found. To contain the difficulties associated with arbitrary parallel and distributed computing, software engineers have begun to develop Concurrent System Architectures (Software Architectures [64] – e.g., layered, object-oriented, implicit invocation, repositories, and interpreters) to provide exactly this kind of guidance and discipline for the design of applications.

In fact, the *pipe-and-filter* system architecture is essentially a process network model. *Filter* (operators, STMs or SFSMs) are threads of computation interconnected by *pipes* (FIFO buffers, streams). This suggests we think about computation as a set of filters that transform input streams to output streams. Pipes provide communication and synchronization. With all communication occurring through pipes, we avoid the need for arbitrary shared memory and the difficulties it poses to achieving atomicity (we need not outlaw using shared memory to efficiently implement pipes in a software multithreaded system – we merely restrict the use of shared memory to a level of the infrastructure where we can use a standard, well-debugged solution rather than leaving it as something every programmer must write and potentially get wrong). If we use a discipline that avoids peeking at present data on the pipes (e.g., avoid unix select-type operations), we can guarantee determinism as noted above.

The software architecture community identifies roughly a dozen distinct concurrent system architectures and hypothesizes that most tasks are naturally captured by one of these architectures. These general architectures represent the things that are common between applications, allowing us to share lessons and infrastructure between very different tasks.

The pipe-and-filter architecture naturally matches the strength of FPGAs identified above (Section 2.1). In particular, SCORE, and pipe-and-filter architectures in general, are good for applications which are compute centric where we can organize the computation as a persistent series of

transforms (operators, filters) with limited state. Most all signal, image, and graphics processing tasks fall into this set, as do many network and text processing tasks including sorting and searching.

SCORE and pipe-and-filter architectures are not appropriate for applications which are data centric; that is, applications which are best modelled as performing operations on a large, evolving database are not naturally captured by these architectures. These data-centric applications would better be captured by transactional or repository models. Identifying how other software architectures can be profitably used as guides for reconfigurable applications and reconfigurable architectures is an important area for future work in order to bring engineering discipline to reconfigurable computing.

2.4. Parallelism

Using this pipe-and-filter architecture, a fine-grained reconfigurable device (e.g., FPGA) can support the four levels of parallelism that may exists in applications.

- **Bit-Level** – operators implemented on FPGAs can exploit gate-level control to implement designs with a minimum number of gates, while exploiting all the parallelism available at the gate level.
- **Instruction-Level** – an operator can perform operations in parallel; tools can use the ILP-extraction techniques developed for processors to synthesize parallel datapaths (e.g., [12,13]).
- **Thread-Level** – each operator (STM, SFSM) is its own thread of control and operates concurrently with all other operators.
- **Data-Level** – within an operator, datapaths can be replicated in parallel to support data-level parallelism; alternately, parallel copies of operators can be instantiated to handle data. In advanced cases, perhaps with more specialized software architectures (e.g., multi-dimensional streams [4], blocked data), it may be possible to automatically split an operator into multiple, data-parallel hardware pipelines (e.g., [32]).

2.5. Virtualization model

Many researchers have proposed Run-Time Reconfiguration (RTR) as a powerful technique to exploit the power and flexibility of reconfigurable platforms (e.g., [30,39,70,50,38]). In particular, RTR can potentially:

- reduce the size of the platform required to solve a phased problem by reconfiguring the platform so that it only holds the datapaths to support each phase independently
- reduce the size of the platform required to solve multirate applications by time-multiplexing the low-throughput portions of the application while keeping the high-throughput portions persistent
- reduce the size of the platform and increase its performance by specializing the implementation to the application data set with run-time instantiation and customization of datapaths
- offer time-space tradeoffs so that large applications can run on smaller hardware, or so that applications with limited throughput requirements can be implemented economically on minimum-sized platforms

In principle, application performance should be higher on a larger platform, e.g., one with more Lookup Tables (LUTs) or more datapaths. Unfortunately, the size of the available platform determines the degree and nature of time multiplexing required of an application. If we design an RTR application for a particular sized platform, its performance will not increase when we run it on a larger platform.

2.5.1. Scalable RTR, separate concerns

By abstracting the size of the platform, SCORE supports scalable RTR. Physical reconfiguration of the device occurs *below* the model level. Consequently, the developer provides a single application description, and it is the responsibility of the compiler and runtime system to automatically schedule the application onto the platform.

Since SCORE allows the computation graph to evolve, it facilitates *logical reconfiguration* of the graph. That is, SCORE allows operator and stream instantiation (e.g., malloc, new, and thread start), and it allows operators to end and streams to close.

SCORE separates logical reconfiguration of the application computation from physical reconfiguration of the device. Logical reconfiguration is visible at the model level, since it is a characteristic of the application. Physical reconfiguration is handled by automated tools, since it is driven by the details of each specific platform.

2.5.2. Virtual co-processor

In SCORE, computation on a processor interacts with the reconfigurable hardware through the virtual operator, unlike instruction augmentation ISA models (e.g., GARP [12], PRISC [63], Chimaera [34]) where the processor interacts with the physical co-processor. In the instruction augmentation model, the processor sends data to the reconfigurable co-processor hardware and reads data from it. Consequently, it does not abstract the number of Reconfigurable Functional Units (RFUs). Minimal designs like PRISC and Chimaera are architected to avoid this problem by outlawing state in the RFU. GARP allows state in the RFU, but this forces it to only allow one function to be active in the RFU for each thread of computation.

In SCORE, the communication is always with the virtual operator rather than the physical operator. Consequently, many operators can be instantiated simultaneously, each holding their own state. One communicates with an operator through the logical stream rather than by doing a read/write to physical registers. For larger platforms, multiple

operators may be co-resident, and this provides a way to address them. For smaller platform, operators may not be co-resident, but this allows one to uniformly address non-co-resident operators and to queue up results until the operators become resident.

3. Challenges

To facilitate scaling, SCORE demands that we push the following tasks to load time and/or run time.

- Reconfiguration
- Scheduling
- Placement
- Routing

Unfortunately, these tasks can be slow using conventional devices and approaches. FPGAs from Xilinx and Altera (e.g., [74,2]) have reconfiguration times in the milliseconds to seconds range. Placement and routing for these devices can easily take minutes to hours or even days. Nonetheless, once motivated by the benefits of scalability and late-bound platform mapping, we see that it is possible to engineer solutions to bring those times down to manageable levels (Sections 5.2.1, 8.2, 8.3, and 8.4).

4. Execution patterns

To support the SCORE model, we employ a set of complementary design patterns – i.e., solutions to commonly recurring problems [25]. These particular patterns are presented here to better illuminate the nature of the SCORE model and our work on its implementation. While these patterns cooperate tightly to support SCORE, they have broad applicability and often show up in the context of other computational models and system architectures.

4.1. Streaming data

Streams represent a logical sequence of communication between a producer and a consumer. STREAMING DATA is a pattern for describing an important property of a computation; it abstracts a connection which might be a group of physical wires in a fully spatial implementation. Capturing the stream connection explicitly in the model allows the compiler and the run-time system to optimize around it (e.g., place the producer and consumer close together, allocate resources for this communication). In models where communication is implicit (e.g., shared memory), the model provides no such guidance to the compiler or run-time. By abstracting the communication into a stream, it can be assigned to a buffer in the case where the producer and consumer are not co-resident (Fig. 1(b)), and it can be assigned to physical networking when the produce and consumer are co-resident (Fig. 1(c)). Further, it can use any number of mechanisms (e.g., shared-bus, packet-switched network, time-multiplexed network, configured links) based on the data rate, predictability, and platform capabilities. Once identified as a data stream, when data must go to or from memory, the platform knows which data to prefetch and how to package data to/from memory.

4.2. Tagged data presence

In the presence of varying implementations, varying parallelism, and data-dependent task execution time, it is not possible to know exactly when data will arrive. Tagging data presence allows the system to indicate dynamically when there is real data present versus when there is no data available. Logically, a stall signal on return data from a cache memory system and pipeline bubbles are specific instances of TAGGED DATA PRESENCE; there they deal with the fact that the return time from a memory depends on the platform (e.g., size and organization of the cache) and the data (e.g., set of prior memory accesses). TAGGED DATA PRESENCE tells a buffer when to store data versus when there is none to store, and it tells an operator when there is data to process. Similarly, TAGGED DATA PRESENCE allows an operator to take more time on a task, perhaps because the platform is serialized (e.g., operator is implemented on a sequential processor) or the particular data item requires more cycles of processing (e.g., handling the infrequent case in more cycles than the common case). TAGGED DATA PRESENCE also provides a mechanism to stall operators during partial reconfigurations or exceptions.

4.3. Back pressure

When operations have data-dependent characteristics such as data-dependent firing or data-dependent timing, an operator may not be able to immediately consume an input from a stream. The operator may be busy processing previous inputs, waiting for a matching input from a different source, or waiting for its downstream operator or FIFO to be ready to take more data. Similarly, a FIFO buffer implementing a stream may be full and unable to take an additional data item. Consequently, it is necessary for operators and streams to prevent an upstream producer from sending more data. BACK PRESSURE is a channel that runs in the opposite direction of data flow telling the upstream producer when the downstream buffer or operator can accept data. Logically BACK PRESSURE is a variant of handshaking acknowledgments.

4.4. Queues with back pressure

When operations have data-dependent characteristics such as data-dependent firing or data-dependent timing and we use BACK PRESSURE to stall upstream producers, the whole system might stall as each individual operator is busy. However, since operators may find themselves busy at different times, this could be inefficient. Placing queues between the operators, we allow a producer to continue to produce data even though a consumer may be busy.

Similarly, if the queue between operators has data in it, this allows a consumer to continue to operate even though the producer is busy performing a long operation or, itself, waiting for an input. Placing queues between operators smooths out local fluctuations in the processing rate of operators, allowing the entire computation to run at the average throughput of the slowest operator rather than the worst-case instantaneous throughput of all operators.

4.5. Buffer lock detection and handling

When operators are allowed to produce or consume data in a data-dependent manner, it is not possible to statically determine bounds on the size of the buffer needed to prevent deadlock of the computation. This may create problems where a task works fine in the abstract compute model with unbounded FIFOs but may lock up when run on a platform with fixed-sized buffers. A solution is to detect the occurrence of a full buffer preventing forward progress and then allocate more capacity to the buffer (e.g., [59]). To retain intermediate results during reallocation, it may be necessary to spill buffer contents to secondary storage (e.g., common memory pool or off-chip DRAM).

4.6. Streaming coprocessor

One powerful way to integrate a concurrent, co-processor with an ISA processor is to provide stream links between the ISA processor and the co-processor. The ISA processor can write data into a stream FIFO to go to the co-processor and read data back from stream FIFOs. This decouples the cycle-by-cycle operation of the co-processor from the processor, abstracting the relative timing of the two units. As noted above (Section 2.5.2), in the case that the physical co-processor can be occupied (e.g., allocated to another operator or task), this abstracts out the co-resident presence of the co-processor. It also allows the number of co-processors to vary with the platform.

4.7. Coarse-grained time multiplexing

When the operator graph is too large for the platform, it is necessary to share the physical hardware in time (See Figs. 1(b) and 4). For a reconfigurable platform, this can be done by changing the configuration over time, to implement the graph in pieces. Reconfiguration, however, can be an expensive operation requiring many cycles. To minimize the overhead cost for reconfiguration, we want to run each operator for many cycles between reconfigurations. In particular, if we can assure that each operation runs for a large number of cycles compared to the reconfiguration time, then we can make the overhead for reconfiguration small ($T_{\text{run-before-reconfig}} \gg T_{\text{config}}$). 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 co-resident, also queue up the results. When the input queue is empty or the output queue is full, we reconfigure to the next set of operators.

For this COARSE-GRAINED TIME MULTIPLEXING to be effective, it must be possible to run an operator for a large number of operations without waiting for other operators. In general, this means that a cycle of operators which take data from each other must not involve non-resident operators. If an operator can only process one or a small number of data items before stalling and waiting on a non-resident operator, then we will have to reconfigure often ($T_{\text{run-before-reconfig}} \ll T_{\text{config}}$); the application will thrash and spend all of its time reconfiguring. This is the similar to virtual memory thrashing when the working set is large compared to the physical memory on a machine. As with virtual memory, the key to reasonable performance is to get the working set to fit on the physical device; here the working set is a cycle of mutually dependent operators. If each cycle is smaller than the platform, then COARSE-GRAINED TIME MULTIPLEXING can work effectively by making sure that each cycle is scheduled onto the platform as a set.

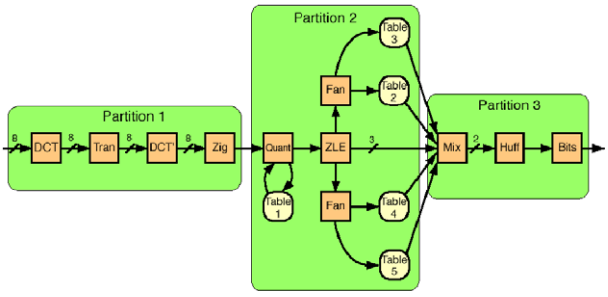

Fig. 4. Partitioning of JPEG image encoder to match platform capacity.

4.8. Fixed-size and standard IO page

As noted above, since the platform size varies, it is necessary to perform placement at load time or run time based on the amount of physical hardware and the time-multiplexed schedule. If we had to place everything at the LUT level, then we would have a very large placement problem. Further, if we allow partial reconfiguration in order to efficiently support the fact that different operators may need to be resident for different amounts of time, we will 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 which have a standard IO 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-sized blocks. This allows us to perform the intra-page placement and routing problem offline at compile time. At runtime, we simply place pages and perform inter-page routing. The runtime placement problem is simplified, since all pages are identically sized and interchangeable. Furthermore, since pages

are typically 100 to 1000 4-LUTs, the run-time placement problem is two to three orders of magnitude smaller than a LUT-level placement. Unfortunately, fixed-size pages may incur internal fragmentation, leaving some resources inside each page unused. Brebner's SLU is an early example of this pattern [10].

Note that this is the same basic idea used in virtual memory. In virtual memory, we do not manage every bit or even every word independently, rather we group a fixed number of words together 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.

4.9. Address generators

When reading through data in memory, it is necessary to: (1) compute the next address, (2) send it to memory, and (3) get data back from the memory. When the latency to memory is long, the round-trip latency to memory can end up limiting the throughput of the application. However, when the set of addresses being accessed is data-independent, it is not necessary for this round-trip latency to impact performance. In particular, if we separate address generation into a separate operator (i.e., thread of control) from data consumption, these two can run independently with arbitrary delay between address generation and data consumption, keeping round-trip latency from impacting throughput. Furthermore, with QUEUES WITH BACK PRESSURE between the address generator, the memory, and the consumer, the queues cover stalls in data access (e.g., on DRAM refresh or row access) and allow operators to run at the average data throughput from memory. For common, special memory patterns (e.g., strided access), the address generator may be implemented in custom hardware as part of the memory controller (e.g., Section 5.2, [9]).

4.10. Sequential vs. parallel

When the platform contains both processors and reconfigurable logic, it is possible to assign some operators to the processor(s) and some to the reconfigurable fabric (this is often called "hardware-software" partitioning; however, since both our spatially parallel reconfigurable fabric and our sequential processor run software, that term can be misleading). We can map operators from our source language either to processor instructions or to reconfigurable implementations, and we can even save both implementations as part of the program executable. At load-time or run-time, low throughput operators can be assigned to the sequential processors, 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 the reconfigurable fabric.

As noted above, COARSE-GRAINED TIME MULTIPLEXING 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. Consequently, one way to get large cycles to fit onto the platform is to push lower throughput operators onto the processor until the cycle is contained.

5. Hardware microarchitecture

A SCORE platform contains compute pages (CPs), configurable memory blocks (CMBs), and a control processor embedded in a uniform, scalable network (Fig. 5).

5.1. Compute pages

Compute Pages (CPs) are the FIXED-SIZED AND STANDARD IO PAGES that contain the reconfigurable fabric elements. Using conventional, LUT-based FPGA elements, a CP might be a modest array containing N 4-LUTs where N is around 100-1000. The CP is the unit of run-time configuration management. In addition to containing the reconfigurable computing units, the CP also contains an interface to the network, including FIFOs to buffer the input streams and perform BACK PRESSURE handshaking with the network (Fig. 6). The CP also contains control logic to control the firing of the reconfigurable datapath based on TAGGED DATA PRESENCE and output BACK PRESSURE, along with control logic for reconfiguration.

In the compute model, CPs perform the same role as operators, communicating via streams and consuming and producing data based on state. The difference is that

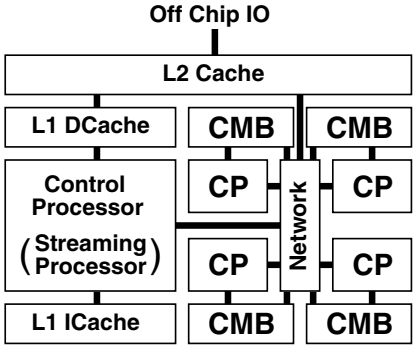

Fig. 5. Illustrative single-chip SCORE platform.

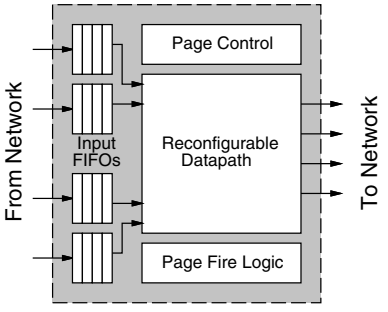

Fig. 6. Compute page (CP) composition.

the CPs are of fixed size to simplify run-time management. The compiler is responsible for mapping between abstract operators with unlimited size and physical operators with finite size; this will include merging multiple, small operators into a single CP as well as splitting large operators into multiple CPs (Section 7.2).

From bottom up implementation consideration, if we make the reconfigurable datapath large enough, the area contribution of the input FIFOs, firing, and control logic will be small compared to the reconfigurable datapath. We expect the area of this control logic to be only a few tens of 4-LUTs, such that compute pages with datapaths on the order of hundreds of 4-LUTs would amortize the control overhead to a modest level.

In concept, CPs can be realized with any kind of programmable or reconfigurable fabric. The visible interaction between CPs and CMBs is the data sent on the connected streams. Consequently, a sequential processor could be a CP. Similarly, a coarse-grained datapath array could be a CP as well (e.g., [18,77,29,55,31,12]). The Application Binary Interface (ABI) for a platform family will need to have compiled versions of each operator for each kind of CP that may appear in the platform family.

5.2. Configurable memory block

A Configurable Memory Block (CMB) is a memory with stream interfaces and a modest amount of local control (Fig. 7). CMBs can hold (1) random access data segments, (2) stream buffers, and (3) state and configurations for CMBs, CPs, and the network. A single CMB may hold many of these things simultaneously based on its capacity (e.g., Fig. 7). However, only one such unit may be active at a time and occupying the CMB's stream interface.

5.2.1. Fast, local reconfiguration

CMBs are physically interleaved with the CPs and the number of CMBs will typically scale with the number of CPs. Consequently, each CP can be associated with a local CMB. When reconfiguration occurs during execution, the CP is loaded from the associated CMB. This provides high bandwidth access to dense configuration memory and allows many CPs to reconfigure in parallel. By scaling the number of CMBs with CPs, reconfiguration time need not increase as the capacity of a SCORE chip grows.

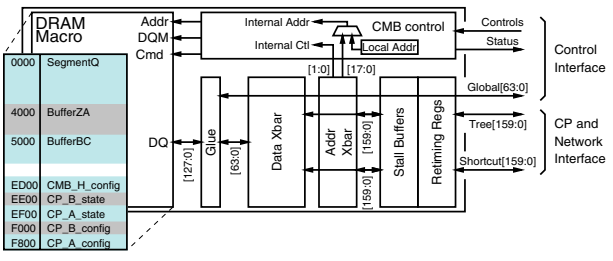

Fig. 7. Configurable memory block (CMB) can hold data, buffer, configuration, and state segments.

We designed a prototype, DRAM-based CMB with a 2 Mb DRAM macro and 2 GB/s of data bandwidth in a 0.4 μm embedded DRAM process [61,60]. The CMB could provide 64b of data every 4 ns cycle. This allowed us to reconfigure a 64 5-LUT CP in less than 350 cycles; that is, less than 1.5 μs. A larger CP would take correspondingly longer; e.g., a 512 5-LUT CP might take around 3000 cycles to reconfigure.

5.2.2. Operating modes

The CMB can be configured as

- random access read-write memory, with stream connections for address, r/w control, data input, and data output
- FIFO, with stream connections for data input and output
- sequential data source
- sequential data sink

Local address logic in the CMB allows it to act as a source, sink, or FIFO without an external address stream. When sourcing CP configurations or storing or loading CP state, the CMB acts as a sequential source or sink on the memory block containing the configuration data or state. As part of the configuration of the active segment, a bounds register tracks the limits of the buffer it is serving. When the bound is reached (e.g., FIFO empty or FIFO full), the CMB asserts BACK PRESSURE to the data streams, and it signals to the control processor that it needs attention. Otherwise, the CMB operates as an independent thread of control like a Direct Memory Access (DMA) engine.

5.2.3. CMB size and balance

As with CPs, by making the memory bank large enough, we can amortize out the interfacing and control overhead. If we then design CPs and CMBs to take up about the same area and build devices with an equal number of CPs and CMBs, we will have a balanced device that is never more than a factor of two away from the optimal ratio of CPs and CMBs. That is, even in unbalanced cases where mostly CPs or mostly CMBs are needed, we never waste more than half of our resources.

5.3. Scalable network

CPs and CMBs are interconnected through a common on-chip network. The network routes data streams which include TAGGED DATA PRESENCE and BACK PRESSURE signals. Conventionally, we consider using a Fat-Tree style configurable network [67,33] which is reconfigured along with CPs and CMBs to provide direct links between producers and consumers. However, the exact nature of the network does not impact the logical computation. The network could use time-multiplexed [23,49] or packet-switched routing [21,51] and could be organized as a mesh (e.g., [6,28]) or Mesh-of-Trees [47,46,27].

5.4. Stream interfacing for processor compute pages

When using a processor as a CP, it is efficient and convenient to augment the processor with streaming operations. To support the streaming operations efficiently, we add a Stream Lookaside Buffer (SLB) (see Fig. 8) which performs a similar role for streams that a TLB performs for virtual memory.

5.4.1. Stream Ops

The two key instructions to add are stream read and stream write.

- STRMWR SRC,SID – Place the value in register SRC onto the stream specified by SID
- STRMRD SID,DST – Read a value from the stream specified by SID and place it in register DST

This allows the processor to transfer data to/from the array in single instructions. With persistent stream links and these instructions added to the processor's ISA, there is no need to spend extra cycles setting up a communication, specifying the destination, or formating a packet (for contrast, compare e.g., [35,20]). This approach integrates streaming communication into the processor abstraction, allowing it to be supported efficiently with hardware.

5.4.2. SLB

On a STRMWR, the processor sends the stream ID and data element to the SLB. The SLB translates the stream to a physical port or a memory address. When the consumer is co-resident in the array, a stream can be configured directly between the CP and the consumer; in this case, if Back Pressure is not asserted, the SLB switches the data item onto the appropriate physical network wires. If Back Pressure is asserted, the processor can be stalled. When the consumer is not co-resident, the SLB will instead turn the operation into a write into the appropriate memory buffer and increment the buffer's tail pointer. Since the streaming operation can be specified with a single instruction, it is possible to provide hardware that handles the operation in a single cycle. In contrast, when the processor must manipulate the buffer pointers for a memory buffer itself, it will usually take several instructions to specify a stream write.

Like a page-table scheme, each process can have its own stream table to provide task isolation. Just as a TLB can enforce access privileges inside an address space, the SLB can enforce permissions on stream access. The SLB will fault to map in streams that have not been loaded or have been evicted due to capacity misses.

Reads have similar mapping to writes, stalling the processor like a memory read when no data is present. If stalled for too long, the scheduler may choose to swap operators; when the operator is mapped back to the processor, it can resume execution by reissuing the stalled stream operation, just as a process that takes a VM page miss will resume by reissuing the faulting memory operation.

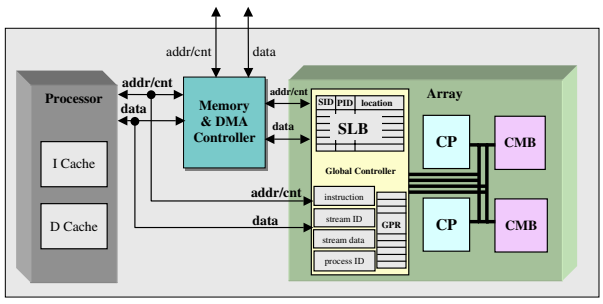

Fig. 8. Processor interface to reconfigurable array including stream interface and array control.

5.5. Control processor/array management

Array reconfiguration is handled by privileged, supervisor mode instructions which instruct the array to stop/start CPs and CMBs and which direct state storage/restoration and configuration loading for CPs, CMBs, and the network. The run-time kernel uses these operations to implement the Coarse-Grained Time Multiplexing. As noted above (Section 5.2.1), the control processor will typically initiate an operation, such as a CP reconfiguration, which will then run autonomously for thousands of cycles. In this way, the control processor can issue a sequence of commands to various CPs and CMBs and have them reconfigure in parallel.

In modest implementations, there may be a single processor to run user operators as well as the run-time kernel. As SCORE chips get larger, it may make sense to dedicate a processor simply to running the kernel scheduler and array reconfiguration. As we scale into the future, there will be a point where it takes more time to issue instructions to initiate reconfigurations from a single control processor than each of the reconfiguration operations; in these cases, it may ultimately make sense to allocate one control processor for every cluster of P CPs and CMBs and to distribute the control among these processors.

6. Design capture

SCORE designs can be captured in a number of ways. For our experimentation, we developed Task Description Format (TDF), a dataflow RTL [16,15]. In hindsight, the key requirements are to capture operators with appropriate dataflow IO interfaces and to allow compositions of operators. One could use SystemC [58] as long as one used a communication library with suitable semantics. Similarly, one could even use Java with a suitable set of class libraries for operators, CMBs, and streams. Other options include YAPI without the probe primitive [22] and Ptolemy designs using domains which provide deterministic process network semantics [42].

7. Compilation

7.1. Operator mapping to FPGAs

We have developed a complete compilation flow from TDF to a commercial FPGA, using Verilog as an intermediate form (Fig. 9) [14]. This flow serves a dual purpose, (1) to map an entire application to a stand-alone FPGA for single context execution, and (2) to evaluate a hypothetical, FPGA-based CP for time-shared execution. The TDF compiler, `tdfc`, automatically generates RTL to efficiently implement the streaming constructs of the TDF language, including flow control checking, stream buffering in queues, and stream pipelining. The compiler then emits a netlist of pages for compilation by a commercial backend. For the data presented in this article, we used Synplify Pro 8.0 and Xilinx ISE 6.3i. Each page contains one or more SFSMs and queues.

We implement a synchronous stream protocol with TAGGED DATA PRESENCE and BACK PRESSURE. A stream is a tuple of unidirectional binary signals (D, E, V, B), where D is the multi-bit *data* payload, V *(valid)* indicates the producer is ready, and B (*backpressure*) indicates the consumer is not ready. At each clock edge, a token of value D is transmitted from producer to consumer if $V \wedge \neg B$. The stream data type is normally extended with an *end of stream* value, encoded out-of-band in signal E, to handle termination conditions. A stream implemented this way may be pipelined in several ways, discussed below.

In an SFSM, each state specifies a set of desired stream inputs and outputs which must be ready before the state action can evaluate, or *fire*. Accordingly, the RTL generated for each state includes a firing guard to check for stream readiness. While the guard is false, the SFSM spins in the same state and emits flow control to stall all streams. When the guard becomes true, the SFSM commits stream production and consumption, evaluates the state action, and transitions to a next state. Fig. 10 shows sample Verilog code generated for an SFSM state. In practice, flow control handling and state actions are generated in separate RTL modules to support separate synthesis. Thus an SFSM is comprised of an FSM module and a datapath module.

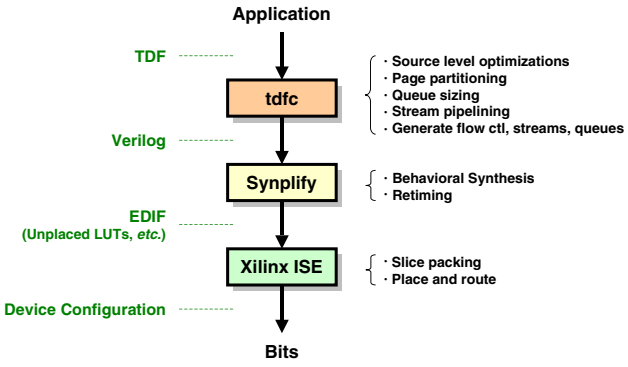

Fig. 9. Compilation flow targeting FPGA.

```
i_b=1; o_v=0; o_e=0;  /* stall */
...
if (state==S) begin
  if (i_v && !i_e && !o_b) begin  /* fire */
    i_b=0; o_v=1; o_e=0; ...
  end
end
```

Fig. 10. Sample, generated RTL (Verilog) for an SFSM state that consumes from stream i and produces to stream o.

Note that an SFSM always waits for incoming flow control before deciding to fire and asserting outgoing flow control. Using this mechanism, two SFSMs cannot be directly connected, since they would wait for each other and deadlock. Instead, SFSMs are separated by assertive, intermediate elements, namely queues.

7.1.1. Inter-operator queuing

One of the key features distinguishing SCORE from RTL models is the use of abstract streams to connect operators. The timing independence of streams allows freedom in the implementation to change the relative timing of operators and of the communication between them. A compiler can exploit that freedom to perform a number of optimizations not possible in a standard RTL model.

Each stream is implemented with one or more queues. Queues are required for correctness of the process network model (Section 2.2). They also abstract communication delays and absorb dynamic variations in consumption/production rates (Section 4.4). Queues serve additional purposes specific to our synthesis methodology, including (1) asserting flow control for the SFSMs at stream endpoints, (2) combinationally decoupling the stream endpoints, (3) pipelining long distance streams, and (4) providing retimable registers to pipeline SFSM datapaths.

A queue may have one of several implementations, depending on the desired capacity and intended purpose. The simplest queue is an enabled register for D, E, and V, with one AND gate for BACK PRESSURE. Despite having only unit capacity, this queue is sufficient for (1), (2), and (4), and it can be cascaded for additional capacity. A queue of medium capacity (for depths up to one to two hundred) can be constructed from shift registers, with data shifting in at one end and out at a dynamic address corresponding to queue occupancy. Our shift register queues use the efficient *SRL16* mode on Xilinx Virtex/Spartan series FPGAs [74,76,75], achieving at least 200MHz (XC2VP70-7) with capacities and bit widths up to 128, e.g., a 16-bit wide, 16-element deep, SRL16 queue requires 56 LUTs, or 28 Virtex-II Slices. Higher capacity queues may be constructed as circular buffers using the embedded RAM on modern FPGAs. Fig. 11 shows our general implementation of a stream using multiple, cascaded queues for different purposes.

When we connect co-resident operators with a stream, we must be careful in selecting the capacity of the stream

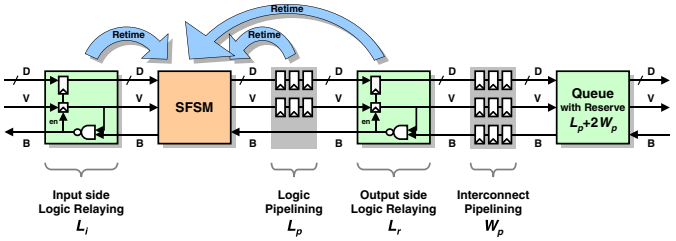

Fig. 11. General, pipelined stream implementation.

queue. To save area, we prefer to minimize the queue capacity. However, a queue with insufficient capacity may create unnecessary stalls in the presence of dynamic data rates or cause the computation to deadlock. As noted in Section 4.5, the general case of queue sizing for process networks with dynamic data consumption and production is undecidable. Nonetheless, in practice, most applications do have meaningful bounds.

Many queue capacities can be bounded by a user annotation or by static analysis. Analysis of queue bounds can be performed using state space exploration, composing automata for SFSMs and queues of particular capacities, and checking for deadlock states [14]. Queues that cannot be statically bounded may still be supported with dynamic allocation using Buffer Lock Detection and Handling.

7.1.2. Interconnect pipelining

Long interconnects will need to be pipelined in order to achieve high throughput operation (e.g., [67,65,69]). We consider two main implementations.

Relaying. Streams spanning long distances may be pipelined by inserting two-element queues to relay the tokens. These two-element queues support registered flow control between operators, to avoid accumulated combinational delays. In the normal case, each relay behaves like a pipeline stage, receiving one input token and sending one output token per clock cycle. When the downstream operator stalls, the two-element queue absorbs the input token like a worm-hole router [57], then asserts B to stall further data from the upstream relay or producer.

Pipelining. Rather than adding relays, we might prefer to add simple registers on all stream wires, including the flow control bits (B,V). Simple registers are much less expensive than two-element queues, both in area and cycle time. However, if we use oblivious registers like this, we must deal with the fact that the flow control bits (B,V) on each side of a W-deep pipelined interconnect become $2W$ cycles stale. That is, if a consumer wishes to stall a producer, it will take W cycles for the consumer's B signal to reach the producer, and when it does, the producer may have placed W additional tokens into the interconnect pipeline. We can accommodate this control staleness by using a Windowed Acknowledgment pattern (e.g., [62]); that is, we modify the downstream queue to assert B when it has $2W$ or fewer empty memory slots.

7.1.3. Operator pipelining

Since SCORE streams abstract the number of clock cycles between operators, we can also accommodate delay for pipelining operator datapaths. The stream implementation allows the addition of registers on the streams, as described above, and subsequent retiming of those registers into the operator logic.

Relaying. We can add an enabled register queue (unit capacity) to a stream and retime its registers into the stream producer or consumer.

Pipelining. Using the Windowed Acknowledgment pattern introduced above, we can add registers with no flow control directly to the forward wires D, E, V, and retime them backwards into the stream producer. We accommodate this as before by modifying the downstream queue to assert B early. Adding L levels of logic pipelining creates L stale flow control bits on the V path, so the downstream queue is modified to assert back-pressure whenever it has L or fewer empty slots. Interconnect pipelining and datapath pipelining may be used together, provided the downstream queue is modified to reserve $2W + L$ empty slots. These stream enabled forms of pipelining are particularly convenient in a synthesis flow since they require no modification of the producer's or consumer's RTL, only a modification of the stream and stream queue.

7.1.4. Mapping results

We have compiled several multimedia applications to the Xilinx Virtex-II Pro XC2VP70 FPGA (speed grade 7) [76]. The applications, listed in Table 1, include an IIR filter, JPEG and wavelet image coders, and an MPEG video coder. They range in size from 8 to 80 SFSMs, and in connectivity from 9 to 238 streams. Mapped to the FPGA, application clock rates range from 87 to 175 MHz. For these results, every stream was implemented identically, using an SRL16-based queue of capacity 16 and pipelining parameters $L_i + L_p + L_r + W_p = 1 + 1 + 0 + 0$ as per Fig. 11. Application area and performance may be improved further by tailoring each stream with its own optimized parameters for queuing and pipelining.

By separately compiling application subcomponents, we can estimate the cost of stream support. Table 1 summarizes those costs for the XC2VP70 target. FSM modules, which contain SFSM firing guards and flow control, comprise only 6.1% of total application area. Stream buffer queues, implemented in Xilinx Slices, comprise 34% of total area. Stream enabled logic pipelining accounts for 7.0% of total area, while providing an average application speedup of 1.5 over unpipelined operators. The remaining area is in SFSM datapaths.

The area for stream buffer queues is non-trivial and thus bears further consideration. Queue area may be reduced by choosing lower queue capacities and by merging connected SFSMs where possible to eliminate queues. Queue area may also be reduced by providing architectural support for queues, e.g., as part of the CP in a SCORE architecture as detailed in Section 5.2.

Table 1
Applications mapped to XC2VP70

Application	SFSMs	Segs	Strms	Clock (MHz)	Area (LUTs)	% Area FSM	% Area queue	% Area pipelining	Speedup pipelining
IIR	8	0	9	175	1984	3.3	27	3.1	1.1
JPEG decode	9	1	50	107	7812	6.6	27	4.7	2.3
JPEG encode	11	4	51	105	7080	7.2	35	5.0	1.8
MPEG encode IP	80	16	238	87	45,602	5.0	36	9.1	1.9
Wavelet encode	30	6	58	132	8580	9.8	31	3.0	1.3
Wavelet decode	27	6	57	127	9112	8.1	28	4.4	1.2
Average						6.1	34	7.0	1.5

7.2. Page mapping

Fig. 12 shows the distribution of SFSM sizes for the operators of the applications used in the previous section. The wide range of operator sizes underscores the need to perform operator packing and splitting in order to target any particular, fixed CP size. We see that most of the operators require less than 512 4-LUTs. This means that page packing will be adequate to reshape most applications. The specific computations which make up the large operators are almost all feed-forward pipelines (e.g., DCT, IDCT) that can be easily decomposed using directional cuts in the dataflow.

For the general case, it will be necessary to decompose large state machines in order to fit them onto small CPs. This could be done by starting with individual states and clustering state logic and datapaths together, obeying the CP area and IO bound. To minimize delay, the goal would be to group together states which typically execute together, so as to minimize the frequency of state transitions that cross the CP boundary. In their work on configuration caching, Li et al. [48] developed a clustering operation which attacks a problem very close to the area- and IO-bound state packing problem that would be necessary for the general CP mapping case.

An important area of future work is to explore the relative benefits of various CP datapath sizes, and a key enabler for that work will be a general operator partitioning scheme. Nonetheless, working with 512+ 4-LUT pages, so far, we have been able to use the simpler, pipeline extraction and dataflow partitioning to decompose operators, so that the page packing problem can be attacked with clustering.

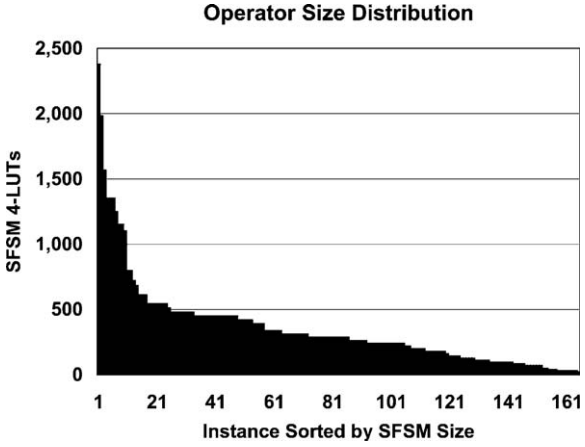

Fig. 12. Operator area distribution.

8. Dynamic runtime support

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 show that these tasks can all be contained to milliseconds, even when using modest platform clock rates of 100 MHz. Further, we show that we can allow this reconfigurability without undermining the isolation and security normally provided by an operating system.

8.1. Operating system security architecture

Since reconfiguration of the physical hardware occurs below the model level, multiple SCORE task graphs can be allocated to the same physical hardware platform and be isolated from each other, just as multiple tasks are isolated from each other on a conventional computer with virtual memory (e.g., multitasking). Operators can only communicate through streams. If an operator does not have a stream link to another operator, it cannot communicate with it. Stream identifiers live in a virtual address space unique to each process, so there is no way to address a stream in another process.

The instructions which reconfigure the array and the SLB are kernel-mode, privileged instructions, just like instructions that modify processor configuration mode registers and TLB contents. Consequently, no user-level operator can directly change the logic configuration on other tasks. Tasks to logically reconfigure the graph (e.g., stream allocation, stream exit, operator instantiation, memory segment instantiation) all go through the operating system the same way that primitive allocations (e.g., sbrk) and process creation/modification/deletion go through the operating system. This can work efficiently, because allocation requests are infrequent compared to use of the allocated resources. Strategic hardware like the SLB avoids the need to have the OS software handle data on a word-by-word basis during operation of an instantiated operator and stream.

8.2. Scheduling

To support SCORE's virtualization model in the presence of late-bound platform mapping, we must have a load-time and run-time 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 until load time. Further, since 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. In order 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, i.e., one that minimizes the time to run the task (minimizes the makespan)
- 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 partition it again. In this way, we amortize the cost of partitioning across the entire application runtime (Fig. 13). If the application will run for seconds, then we can afford tens of milliseconds for this scheduling operation while keeping the overhead small. If we can make the scheduling time smaller, 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 re-partition and re-schedule the graph during execution. The smaller we can make the partitioning time, the more frequently we can afford to invoke the partitioner without paying a large overhead.

In Section 5.2.1 we showed how architectural support could bring configuration down to a few thousand cycles. To fully exploit this fast reconfiguration time, the reconfiguration management time for swapping pages and advancing the schedule should be in the same ballpark. The combination of reconfiguration time and schedule management time determine the duration each configuration should run in order to avoid thrashing (Fig. 13):

$$T_{\text{run-before-reconfig}} \ll (T_{\text{config}} + T_{\text{advance\_schedule}})$$

We have developed a series of schedulers to address these issues, as previously reported [19,53,52]. In this section, we highlight the lessons from those exercises.

8.2.1. Partitioning and scheduling

To fit a large SCORE graph onto a smaller platform, we must first temporally partition the graph into subgraphs that fit on the device. Once the partition is created, the scheduler adds a buffer for each stream that crosses a temporal reconfiguration boundary. Within the temporal partitions, the schedule must also assign buffers to physical resources. As addressed in the next section (Section 8.3), the run-time system will then place the CPs and CMBs in each partition. The results of the partitioning and assignment can be visualized as a schedule table. Table 2 shows what this table looks like for the partitioned JPEG application shown in Fig. 4.

To compute schedules quickly, we use a heuristic based on graph topology. It yields makespans that are within 17% of the optimal single-appearance schedule [53], yet it consumes only 300–700 thousand processor cycles (measured on a Pentium III using on-chip time-stamp counter; see Table 3). Even on a 100 MHz processor, this means schedule generation time would be only 3–7 ms.

The scheduler uses feedback from previous runs to estimate operator production/consumption rates to compute high quality schedules. For a given data set, operator production/consumption rates do not change when the size of

Table 2
Schedule table for JPEG image encoding on a SCORE platform with 4 CPs and 16 CMBs

Resource	Time slice		
	1	2	3···
CP 0	DCT	Fan	Bits
CP 1	Tran	Fan	Huff
CP 2	DCT'	ZLE	Mix
CP 3	Zig	Quant	—
CMB 0	Stitch[1,0]	Stitch[1,0]	Stitch[3,0]
CMB 1	Stitch[0,0]	Stitch[2,0]	Stitch[2,0]
CMB 2	Stitch[0,1]	Table 1	—
CMB 3	Stitch[0,2]	Table 2	—
⋮			

Table 3
Schedule generator load-time overhead (millions of cycles): application-specific costs for three device sizes: minimum feasible (min), quarter spatial (25%), half spatial (50%)

App name	|V|	|E|	Min size	Min	25%	50%
JPEG encode	20	54	4	0.6	0.7	0.7
JPEG decode	16	56	3	0.3	0.3	0.3
Wavelet encode	36	56	6	0.5	0.5	0.5
Wavelet decode	33	51	6	0.3	0.3	0.3

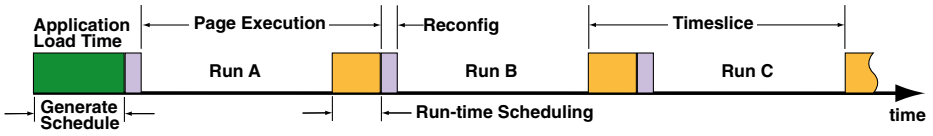

Fig. 13. Application execution timeline.

the platform changes. If the production and consumption rates are consistent across data sets, then this can be a good approximation of operator behavior.

8.2.2. Advancing the schedule

During execution, the scheduler must reconfigure the device and handle dynamic events. In the simplest case, we employ a *static* schedule that simply allows each partition to run for a specified length of time, the *timeslice*. At the end of the timeslice, the scheduler will perform BUFFER LOCK DETECTION and HANDLING (Section 4.5). We were able to design this simple, static case so that the scheduler handling only required about ten thousand cycles per timeslice (See Fig. 14).

A potential weakness of a strict, timeslice-based, static scheduler is that it is oblivious to dynamics in production/consumption rates. If input buffers drain or output buffers fill at an unexpected rate, all resident operators may stall before the end of the timeslice, resulting in significant under-utilization of the hardware.

This inspired us to create a *quasi-static* scheduler to improve schedule quality. The quasi-static scheduler adds a small amount of additional hardware that can detect when the resident pages are all stalled. When this dynamic stall event is detected, the current timeslice is immediately terminated, triggering BUFFER LOCK DETECTION and HANDLING and reconfiguration. By simply watching when all CPs and CMBs are stalled for a number of successive cycles that exceeds the network latency, we know that data has drained from the network, and no more operations can be performed in this timeslice. This simple and inexpensive detection scheme does cause us to lose tens of cycles during the network drain and detection, but since that delay is small compared to the reconfiguration time, it has minor impact on overall performance.

In the extreme, we could use a fully *dynamic* scheduler. The dynamic scheduler can decide independently which operators should be run in each timeslice. Here, we have the potential to look at fullness and emptiness of queues and schedule the operators that will run for the longest amounts of time. Our dynamic scheduler [19] used a *priority-list* scheduler to handle data-dependent variations in token flow rates; node priorities were based on input token and output buffer space availability. The computation time associated with each reconfiguration was quite high, on the order of 50,000–150,000 cycles (See Fig. 14) – almost an order of magnitude higher than the handling required by the static and quasi-static schedulers.

8.2.3. Schedule quality

Fig. 15 shows the total execution time of the wavelet-based encoder application for various array sizes, using the dynamic, static, and quasi-static schedulers. This encoder application requires 30 physical pages for a fully spatial implementation. The graph vertical axis shows the total time to compress a 512×512 image. The horizontal axis shows the device size in compute page and configurable memory block pairs. Performance was measured on a cycle-level fabric simulator.

All curves exhibit the expected performance scaling behavior: more hardware results in an equal or lower execution time. Minor non-monotonicity with respect to the device size is due to anomalous scheduler effects. The static and quasi-static curves demonstrate that the static schedulers yields higher quality schedules than the dynamic approach. The execution time reduction from the dynamic to quasi-static approach is a factor of four on average. Comparing execution times for the quasi-static and static schedulers, we find that the stall-detect feature contributes on average a factor of two in improved performance.

The static schedulers outperform the dynamic scheduler, because they evaluate the graph globally rather than performing greedy graph analysis. The perceived advantages of the fully dynamic scheduler, such as its ability to adapt its scheduling decisions to match data-flow patterns, are not realized at a feasible scheduling granularity. Its timeslice size is constrained by the large scheduling overhead of 50–150 thousand cycles per time-slice.

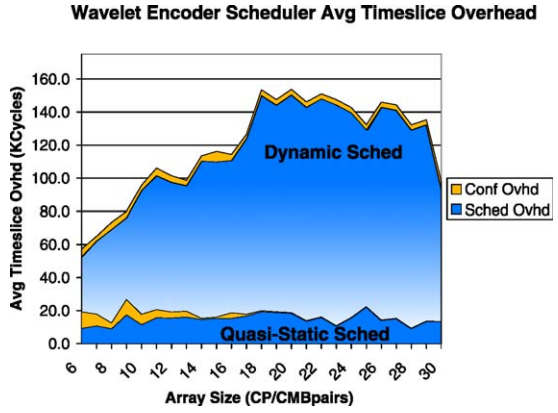

Fig. 14. Wavelet encoder: comparison of average timeslice overhead.

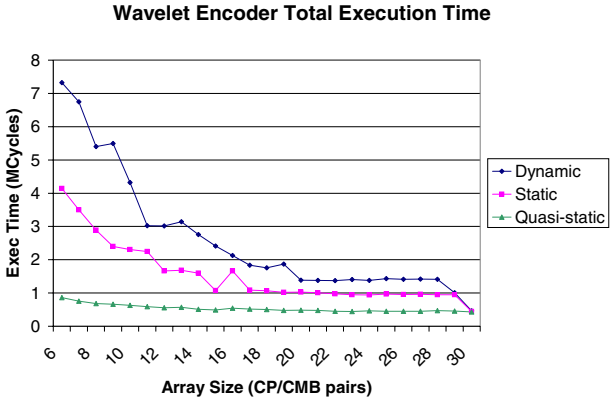

Fig. 15. Wavelet encoder: comparison of total application execution time under different schedulers.

8.2.4. Summary of results

The schedulers were evaluated with JPEG and wavelet-based codecs [52], which represent a typical workload for a SCORE platform. The applications combine static-rate operators and data-dependent, dynamic-rate operators. Table 4 shows that the quasi-static scheduler reduces the fraction of the execution time attributable to scheduling overhead from 30–40% down to 5–10% relative to the dynamic scheduler. Application performance improves under the quasi-static scheduler by around 3–5×. Notice that the speedups are greater on smaller reconfigurable devices with limited resources, where the improved scheduling quality and efficiency are more critical.

The quasi-static scheduling strategy reaches key performance objectives without restricting the dynamic computation model of SCORE. Inner scheduling loop overhead is contained under ten thousand processor cycles per time-slice (100 μs on 100 MHz core), allowing the scheduler to react quickly to changes in a computation. The load-time overhead is contained under 700,000 cycles (7 ms). The quasi-static scheduler achieves the highest scheduling quality and device utilization through simple yet effective scheduling heuristics and responds to the gross dynamic behavior of the application using simple hardware feedback.

Table 4
Execution time and scheduling overhead summary: execution time reported in millions of cycles

Array size	Dynamic		Quasi-static		Speedup
	Exec time	% Ovhd	Exec time	% Ovhd	
Wavelet encoder (30 pages)					
6	7.324	*18.5*	0.859	7.5	**8.52**
8	5.400	*22.9*	0.683	*9.3*	**7.91**
14	2.754	*32.0*	0.513	*9.3*	**5.37**
18	1.757	*38.2*	0.503	*8.6*	**3.50**
24	1.378	*43.7*	0.461	*11.8*	**2.99**
26	1.410	*45.1*	0.453	*8.9*	**3.12**
Wavelet decoder (30 pages)					
6	11.631	*17.0*	0.846	*4.0*	**13.75**
8	9.137	*19.0*	0.753	*4.3*	**12.13**
14	6.887	*25.6*	0.553	*4.1*	**12.46**
18	2.803	*29.0*	0.534	*4.3*	**5.25**
24	1.717	*34.4*	0.520	*4.1*	**3.30**
26	1.167	*36.7*	0.539	*4.0*	**2.17**
JPEG encoder (13 pages)					
4	6.368	*24.1*	2.341	*7.3*	**2.72**
5	9.086	*21.5*	1.479	*5.3*	**6.15**
8	2.349	*29.6*	1.360	*6.0*	**1.73**
9	2.406	*30.5*	1.052	*3.8*	**2.29**
12	3.678	*28.7*	0.991	*3.3*	**3.71**
13	0.939	*15.1*	0.879	*1.6*	**1.07**
JPEG decoder (12 pages)					
3	13.312	*19.8*	3.004	*5.6*	**4.43**
4	8.010	*26.0*	1.701	*7.1*	**4.71**
7	2.589	*28.2*	1.192	*3.3*	**2.17**
8	2.196	*24.8*	0.979	*3.4*	**2.24**
11	2.518	*35.5*	0.965	*3.4*	**2.61**
12	0.910	*12.6*	0.869	*1.5*	**1.05**

8.3. Placement

Using the FIXED-SIZED and STANDARD IO PAGES pattern introduced above, we immediately reduce the magnitude of the placement task by 2–3 orders of magnitude. Nonetheless, the placement task may still take too long when run using conventional single-processor-based placers for reconfiguration-time or even load-time placement. Fortunately, once we have a spatially parallel reconfigurable computing platform, we can use the reconfigurable platform itself to perform placement substantially faster. In [72,71], we show how to perform simulated annealing spatially with reconfigurable logic; we can place a graph with 1000 movable elements in roughly one million cycles. Even if we only ran the placement engine at 100 MHz, this would mean we can perform placement in 10 ms. If each CP held 512 4-LUTs, then 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 the CP large enough, then each CP can act as a systolic placement cell. As a placement cell, it holds a candidate, logical page and negotiates exchanges with its nearest neighbors. A pair of adjacent CPs 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 CPs can be paired up and negotiate swaps in parallel, allowing many moves per swap epoch. By pairing up only neighbors, we can guarantee to require minimal interconnect for this placement engine and keep the cycle times short. Since there is one systolic placement cell for every page site on the device, the hardware and parallelism in the placement engine scale exactly as the size of the placement problem that needs to be solved. In [72], we estimate that 400 4-LUTs is adequate to implement a 100 MHz systolic placement cell on Xilinx Virtex-II [73] generation hardware; this suggests that SCORE platforms with 512 LUT CPs will certainly be able to perform their own placement.

8.4. Routing

Once the pages have been placed, we must perform inter-page routing. Again, we can exploit the fact that we have a spatially parallel computing platform to route tasks in 100,000 to a million cycles (Table 5) [26]; that is, the

Table 5
Spatial routing for SCORE applications (from [26])

Application	Compute pages	Array size	IOs	Channel width	Cycles
MPEG encode	92	128	18	25	110,567
JPEG encode	13	16	16	18	21,917
JPEG decode	12	16	16	18	17,186
Wavelet encode	30	32	6	8	16,118

route task can be performed in 1–10 ms assuming only 100 MHz operation. Here, we augment the inter-page network with additional logic to allow the network itself to identify all free paths between a source node and a sink node in parallel. This allows a flooding search 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 [54], the state-of-the-art software routing algorithm for FPGAs [37]. Using word-wide (e.g., 16-bit) datapaths for the inter-page network, the additional area overhead for this augmented network is less than 30% 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 inter-page routing (e.g., [51]) to avoid the need to compute and configure the network. Packet switches are generally much larger and higher latency than configured switches, but they may be able to handle multirate and dynamic traffic more efficiently.

9. Lessons

For reconfigurable computing systems to fully deliver their potential as flexible, programmable systems with high computational capacity, we must develop suitable models for developing scalable and reusable applications and application components. As such, we need models that provide stable interfaces for software that abstract out cycle-by-cycle timing, platform capacity, and microarchitectural details of the computing fabric. Further, to make it possible for a wider range of developers to understand and exploit reconfigurable computing platforms, we need system architectures that embody our best practices and understanding of how to properly organize and manage computations for these platforms.

SCORE provides one such model. It guides developers to an efficient pipe-and-filter organization which exploits the ability of reconfigurable fabrics to efficiently support pipelined datapaths that perform the same computation repeatedly at high throughput. It provides a timing-independent model to allow automated scaling of the platform hardware. In so doing, it tackles system-level issues, including runtime management and allocation, which are not standardized when one simply deals with raw hardware.

We have shown that we can support the SCORE model efficiently with suitable execution patterns, hardware support, and lightweight runtime software. We describe these patterns here in the context of the SCORE model, but they are by no means unique to SCORE and are useful in other contexts. We can engineer reconfiguration to occur in microseconds, and we can develop quasi-static schedulers that drive reconfiguration in microseconds. For late-bound platform mapping, we can schedule computations to the platform in milliseconds and use the device itself to compute placements and routes in milliseconds. Using coarse-grained pages (512 4-LUT capacity or larger), the added logic overhead to support the SCORE model can be modest (10–20%).

While virtualization has been a key focus in the design of SCORE, the SCORE model proves useful even when designing static, spatial designs for conventional FPGAs [14]. As suggested in Section 7.1, the model allows us to separately compile operators and to pipeline both operators and interconnect. One could use the FIXED-SIZED AND STANDARD IO PAGES pattern as a discipline to speed design assembly and mapping to FPGAs.

10. Future directions

The abstractions and mechanisms reviewed and developed in the article do have their limitations, and those limitations suggest some important directions for further development.

10.1. Platform dependent graph size

What happens once the platform can implement a graph fully spatially? For many of the applications considered, the data is blocked, and it would be possible to exploit even further parallelism by instantiating parallel data paths (e.g., different datapaths for alternate rows or blocks in an image compression). One could simply build this larger, more parallel operation graph, but it is then less efficient to time-multiplex this large graph onto smaller hardware. Ideally, we should generate the appropriate graph structure based on load-time information about the size of the platform. This requires a richer run-time model and the ability to algorithmically capture the freedom and alternatives for implementing operators. With the inexpensive reconfiguration, scheduling, placement, and routing developed to enable SCORE, it is viable to consider this kind of run-time graph construction.

10.2. Alternate execution patterns

While COARSE-GRAINED TIME-MULTIPLEXING and SEQUENTIAL VS. PARALLEL are powerful patterns for scaling, if we really want to scale across the orders of magnitude more device capacity that we will see in the next several decades, it is clear that we need to enable an even larger set of techniques for scaling. Further, if we can more efficiently handle large graphs on modest platforms, we can expand the set of applications where the SCORE model is viable. Techniques such as FINE-GRAINED TIME MULTIPLEXING and COMMON OPERATING SHARING might allow modest platforms to run graphs with large cyclic dependencies without thrashing.

Future runtime systems might choose between the execution pattern based on the characteristic of the application graph and platform.

10.3. Develop reconfigurable system architectures

The pipe-and-filter system architecture is not the natural or appropriate way to implement many computations. For example, data-centric operations are more naturally viewed as *repositories* [64]. Consequently, we need to explore and develop a set of reconfigurable system architectures which cover the space of applications which reconfigurable computing systems can implement efficiently. These architectures can be the basis for developing compilation tools and run-time support to ease the task of developing reconfigurable solutions. The architecture disciplines will further serve as the basis for guiding designers to good reconfigurable application design, building on the growing body of engineering knowledge.

Acknowledgements

This work was supported by the Defense Advanced Research Projects Agency under contract number DABT63-C-0048, NSF CAREER program under Grant CCR-0133102, NSF ITR program under grant number CCR-0205471, the California MICRO Program, and Xilinx.

References

[1] M.S. Vikas Agarwal, Hrishikesh, Stephen W. Keckler, Doug Burger, Clock rate versus ipc: the end of the road for conventional microarchitectures, in: Proceedings of the International Symposium on Computer Architecture, 2000, pp. 248–259.
[2] Altera Corporation, 2610 Orchard Parkway, San Jose, CA 95134-2020. Stratix II Device Handbook, 4.0 edition, December 2005.
[3] G.M. Amdahl, G.A. Blaauw, Frederick Jr., P. Brooks, Architecture of the IBM system/360, IBM Journal of Research and Development (1964) 87–101.
[4] E.A. Ashchroft, A.A. Faustini, R. Jagannathan, W.W. Wadge, Multidimensional Programming, Oxford University Press, Oxford, 1995.
[5] K. Bazargan, R. Kastner, M. Sarrafzadeh, Fast template placement for reconfigurable computing systems, IEEE Design and Test of Computers 17 (1) (2000) 68–83.
[6] Vaughn Betz, Jonathan Rose, Alexander Marquardt. Architecture and CAD for Deep-Submicron FPGAs. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts, 02061 USA, 1999.
[7] S.S. Bhattacharyya, P.K. Murthy, E.A. Lee, Software Synthesis from Dataflow Graphs, chapter Synchronous Dataflow, Kluwer Academic Publishers, 1996.
[8] G. Bilsen, M. Engels, R. Lauwereins, J. Peperstraete, Cyclo-static dataflow, IEEE Transactions on Signal Processing 44 (2) (1996) 397–408.
[9] V. Michael Bove, Jr., Mark Lee, Christopher McEniry, Thomas Nwodoh, John Watlington, Media processing with field-programmable gate arrays on a microprocessor's local bus, in: Proceedings of SPIE Media Processors, vol. 3655, 1999.
[10] Gordon Brebner, The swapable logic unit: a paradigm for virtual hardware, in: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, April 1997, pp. 77–86.
[11] Joseph T. Buck, Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model, PhD thesis, University of California, Berkeley, 1993. ERL Technical Report 93/69.
[12] T.J. Callahan, J. Hauser, J. Wawrzynek, The Garp architecture and C compiler, IEEE Computer 33 (4) (2000) 62–69.
[13] T.J. Callahan, J. Wawrzynek, Instruction level parallelism for reconfigurable computing, in: Hartenstein, Keevallik (Eds.), Proceedings of the Eighth International Workshop on Field Programmable Logic and Applications, Springer-Verlag, 1998.
[14] Eylon Caspi, Design Automation for Streaming Systems, Ph.D. Thesis, EECS Department, University of California, Berkeley, December 16, 2005.
[15] Eylon Caspi, Programming SCORE, Technical Report UCB/EECS-2005-25, EECS Department, University of California, Berkeley, December 16, 2005.
[16] Eylon Caspi, Michael Chu, Randy Huang, Nicholas Weaver, Joseph Yeh, John Wawrzynek, André DeHon, Stream Computations Organized for Reconfigurable Execution (SCORE): Introduction and Tutorial. <http://www.cs.berkeley.edu/projects/brass/documents/score_tutorial.html>, short version appears in FPL'2000 (LNCS 1896), 2000.
[17] E. Caspi, M. Chu, R. Huang, N. Weaver, J. Yeh, J. Wawrzynek, A. DeHon, Stream computations organized for reconfigurable execution (SCORE): extended abstract, in: Proceedings of the International Conference on Field-Programmable Logic and Applications LNCS, Springer-Verlag, 2000, pp. 605–614.
[18] D.C. Chen, J.M. Rabaey, A reconfigurable multiprocessor ic for rapid prototyping of algorithmic-specific high-speed dsp data paths, IEEE Journal of Solid-State Circuits 27 (12) (1992) 1895–1904.
[19] Michael Chu. Dynamic Runtime Scheduler Support for SCORE. Master's thesis, University of California, Berkeley, December 2000.
[20] W.J. Dally, S.J.A. Fiske, J.S. Keen, R.A. Lethin, M.D. Noakes, P.R. Nuth, R.E. Davison, G.A. Fyler, The message-driven processor: A multicomputer processing node with efficient mechanisms, IEEE Micro (1992) 23–39.
[21] William J. Dally, Brian Towles. Route packets, not wires: on-chip interconnection networks, in: Design Automation Conference, 2001, pp. 684–689.
[22] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, K.A. Vissers. Application modeling for signal processing systems, in: Proceedings of the ACM/IEEE Design Automation Conference, 2000, pp. 402–405.
[23] André DeHon. Reconfigurable Architectures for General-Purpose Computing. AI Technical Report 1586, MIT Artificial Intelligence Laboratory, 545 Technology Sq., Cambridge, MA 02139, October 1996.
[24] A. DeHon, The density advantage of configurable computing, IEEE Computer 33 (4) (2000) 41–49.
[25] André DeHon, Joshua Adams, Michael deLorimier, Nachiket Kapre, Yuki Matsuda, Helia Naeimi, Michael Vanier, Michael Wrighton, Design Patterns for Reconfigurable Computing, in: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, April 2004.
[26] André DeHon, Randy Huang, John Wawrzynek, Hardware-assisted fast routing, in: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, April 2002, pp. 205–215.
[27] A. DeHon, R. Rubin, Design of FPGA interconnect for multilevel metalization, IEEE Transactions on VLSI Systems 12 (10) (2004) 1038–1050.
[28] J. Duato, S. Yalamanchili, L. Ni, Interconnection Networks: An Engineering Approach, Morgan Kaufmann Publishers, San Francisco, 2003.
[29] C. Ebeling, D. Cronquist, P. Franklin, Rapid – reconfigurable pipelined datapath, in: Proceedings of the International Conference on Field-Programmable Logic and ApplicationsLecture Notes in Computer Science, 1142, Springer, 1996, pp. 126–135.

[30] James G. Eldredge, Brad L. Hutchings, Density enhancement of a neural network using FPGAs and run-time reconfiguration, in: Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, April 1994, pp. 180–188.
[31] Seth C. Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, Ronald Laufer, Piperench: a coprocessor for streaming multimedia acceleration, in: Proceedings of the International Symposium on Computer Architecture, May 1999, 28–39.
[32] Michael Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Christopher Leger, Andrew A. Lamb, Jeremy Wong, Henry Hoffman, David Z. Maze, Saman Amarasinghe, A stream compiler for communication-exposed architectures, in: International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002.
[33] Ronald I. Greenberg, Charles E. Leiserson, Randomness in Computation, vol. 5 of Advances in Computing Research, chapter Randomized Routing on Fat-Trees, JAI Press, 1988. Earlier version MIT/LCS/TM-307.
[34] Scott Hauck, Thomas Fry, Matthew Hosler, Jeffery Kao, The chimaera reconfigurable functional unit, in: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, April 1997, 87–96.
[35] Dana S. Henry, Christopher F. Joerg, A tightly-coupled processor-network interface, in: Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 1992.
[36] C.A.R. Hoare, Communicating sequential processes, in: International Series in Computer Science, Prentice-Hall, Englewood Cliffs, NJ, 1985.
[37] Randy Huang, John Wawrzynek, André DeHon, stochastic, spatial routing for hypergraphs, trees, and meshes, in: Proceedings of the International Symposium on Field-Programmable Gate Arrays, February 2003, pp. 78–87.
[38] Rhett D. Hudson, David I. Lehn, Peter M. Athanas, A run-time reconfigurable engine for image interpolation, in: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, 1998, pp. 88–95.
[39] Chris Jones, John Oswald, Brian Schoner, John Villasenor, Issues in wireless video coding using run-time-reconfigurable fpgas, in: Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, April 1995, 85–89.
[40] G. Kahn, The semantics of a simple language for parallel programming, in: Proceedings of the IFIP CONGRESS H, North-Holland Publishing Company, Amsterdam, 1974, pp. 471–475.
[41] G. Kahn, D.B. MacQueen, Coroutines and networks of parallel processes, in: Proceedings of the IFIP CONGRESS 77, North-Holland Publishing Company, Amsterdam, 1977, pp. 993–998.
[42] Edward Lee, UC Berkeley ptolemy project. Online: <http://www.ptolemy.eecs.berkeley.edu/>, 2005.
[43] E.A. Lee, S. Neuendorffer, Concurrent models of computation for embedded software, IEE Proceedings-Computers and Digital Techniques 152 (2) (2005) 239–250.
[44] E.A. Lee, A. Sangiovanni-Vincentelli, A framework for comparing models of computation, IEEE Transactions on Computed-Aided Design for Integrated Circuits and Systems 17 (12) (1998) 1217–1229.
[45] E.A. Lee, Advanced Topics in Dataflow Computing, chapter Static Scheduling of Data-Flow Programs for DSP, Prentice-Hall, Englewood Cliffs, NJ, 1991.
[46] F.T. Leighton, New lower bound techniques for VLSI, in: Twenty-Second Annual Symposium on the Foundations of Computer Science, IEEE, 1981, pp. 1–12.
[47] F.T. Leighton, Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1992.
[48] Z. Li, K. Compton, S. Hauck, Configuration caching techniques for FPGA, in: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, IEEE, 2000.
[49] J. Liang, A. Laffely, S. Srinivasan, R. Tessier, An architecture and compiler for scalable on-chip communication, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 12 (7) (2004) 711–726.
[50] Wayne Luk, Nabeel Shirazi, Peter Y.K. Cheung, Compilation tools for run-time reconfigurable designs, in: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, April 1997, pp. 56–65.
[51] T. Marescaux, V. Nollet, J.-Y. Mignolet, A.B.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 (l) (2004) 107–130.
[52] Yury Markovskiy, Quasi-Static Scheduling for SCORE. Master's thesis, University of California, Berkeley, December 2004.
[53] Yury Markovskiy, Eylon Caspi, Randy Huang, Joseph Yeh, Michael Chu, John Wawrzynek, André DeHon, Analysis of quasistatic scheduling techniques in a virtualized reconfigurable machine, in: Proceedings of the International Symposium on Field-Programmable Gate Arrays, February 2002, 196–205.
[54] Larry McMurchie, Carl Ebling, PathFinder: a negotiation-based performance-driven router for FPGAs, in: Proceedings of the International Symposium on Field-Programmable Gate Arrays, ACM, February 1995, pp. 111–117 .
[55] Ethan Mirsky, André DeHon, MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources, in Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, April 1996.
[56] Bruce Newgard, Signal processing with Xilinx FPGAs. <http://www.xilinx.com/apps/apprLOtes/sd_xdsp.pdf>, June 1996.
[57] L.M. Ni, P.K. McKinley, A survey of wormhole routing techniques in direct networks, IEEE Computer (1993).
[58] Open System C Initiative, <http://www.systemc.org>. System C 2.1 Language Reference Manual, May 2005.
[59] Thomas M. Parks, Bounded Scheduling of Process Networks. UCB/ERL 95-105, University of California, Berkeley, 1995.
[60] Stylianos Perissakis, Balancing Computation and Memory in High Capacity Reconfigurable Arrays, Ph.D. Thesis, University of California, Berkeley, 2000.
[61] Stylianos Perissakis, Yangsung Joo, Jinhong Ahn, André DeHon, John Wawrzynek, Embedded DRAM for a reconfigurable array, in Proceedings of the 1999 Symposium on VLSI Circuits, June 1999.
[62] Jon Postel. Transmission Control Protocol – DARPA Internet Program Protocol Specification. RFC 793, USC/ISI, Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, California, 90291, September 1981.
[63] R. Razdan, M.D. Smith, A high-performance microarchitecture with hardware-programmable functional units, in: Proceedings of the 27th Annual International Symposium on Microarchitecture, IEEE Computer Society, 1994, pp. 172–180.
[64] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall, Englewood Cliffs, NJ, 1996.
[65] Deshanand P. Singh,. Stephen D. Brown, The case for registered routing switches in field programmable gate arrays, in: Proceedings of the International Symposium on Field-Programmable Gate Arrays, February 2001, pp. 161–169.
[66] D. Sylvester, K. Keutzer, Rethinking deep-submicron circuit design, IEEE Computer 32 (11) (1999) 25–33.
[67] William Tsu, Kip Macy, Atul Joshi, Randy Huang, Norman Walker, Tony Tung, Omid Rowhani, Varghese George, John Wawrzynek, André DeHon, HSRA: high-speed, hierarchical synchronous reconfigurable array, in: Proceedings of the International Symposium on Field-Programmable Gate Arrays, February 1999, pp. 125–134.
[68] J. Villasenor, B. Schoner, K.-N. Chia, C. Zapata, Configurable computer solutions for automatic target recognition, in: Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, IEEE, 1996, pp. 70–79.

[69] Nicholas Weaver, John Hauser, John Wawrzynek, The SFRA: a corner-turn FPGA architecture, in: Proceedings of the International Symposium on Field-Programmable Gate Arrays, March 2004.
[70] Michael J. Wirthlin, Brad L. Hutchings, Sequencing run-time reconfigured hardware with software, in: Proceedings of the International Symposium on Field Programmable Gate Arrays, February 1996, pp. 122–128.
[71] Michael Wrighton, A Spatial Approach to FPGA Cell Placement by Simulated Annealing, Master's thesis, California Institute of Technology, June 2003.
[72] Michael Wrighton, André DeHon, Hardware-assisted simulated annealing with application for fast FPGA placement, in: Proceedings of the International Symposium on Field-Programmable Gate Arrays, February 2003, pp. 122–128.
[73] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. Xilinx Virtex-II 1.5V Platform FPGAs Data Sheet, July 2002. DS031 <http://www.xilinx.com/partinfo/ds031.pdf>.
[74] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. Xilinx Virtex-II Platform FPGAs Data Sheet, October 2003. DS031 <http://direct.xilinx.com/bvdocs/publications/ds031.pdf>.
[75] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. Xilinx Spartan-3 FPGA Family Data Sheet, December 2004. DS099 <http://direct.xilinx.com/bvdocs/publications/ds099.pdf>.
[76] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. Xilinx Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet, October 2005. DS083 <http://direct.xilinx.com/bvdocs/publications/ds083.pdf>.
[77] A.K. Yeung, J.M. Rabaey, A 2.4 gops data-drivern reconfigurable multiprocessor ic for dsp, in: Proceedings of the 1995 IEEE International Solid-State Circuits Conference, IEEE, 1995, pp. 108–109.
[78] Peixin Zhong, Margaret Martonosi, Pranav Ashar, Sharad Malik, Accelerating boolean satisfiability with configurable hardware, in: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, April 1998, pp. 186–195.