Stream Computations Organized for Reconfigurable Execution (SCORE): Introduction and Tutorial

Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, Yury Markovskiy
André DeHon, John Wawrzynek
U.C. Berkeley BRASS group
<andre@acm.org>

August 25, 2000 / Version 1.0

Abstract

A primary impediment to wide-spread exploitation of reconfigurable computing is the lack of a unifying computational model which allows application portability and longevity without sacrificing a substantial fraction of the raw capabilities. We introduce SCORE (Stream Computation Organized for Reconfigurable Execution), a stream-based compute model which virtualizes reconfigurable computing resources (compute, storage, and communication) by dividing a computation up into fixed-size "pages" and time-multiplexing the virtual pages on available physical hardware. Consequently, SCORE applications can scale up or down automatically to exploit a wide range of hardware sizes. We hypothesize that the SCORE model will ease development and deployment of reconfigurable applications and expand the range of applications which can benefit from reconfigurable execution. Further, we believe that a well engineered SCORE implementation can be efficient, wasting little of the capabilities of the raw hardware. In this paper, we introduce the key components of the SCORE system.

1 Introduction

A large body of evidence exists documenting the raw advantages of reconfigurable hardware such as FPGAs over conventional microprocessor-based systems on selected applications. Yet reconfigurable computing remains in limited use, popular primarily in application-specific domains (*e.g.* [23] [32] [38]) or as a replacement for ASICs for rapid prototyping and fast time-to-market. This limited popularity is not due to any lack of raw hardware capability, as million-gate devices are readily available [37] [2], and we have seen recent advances in high clock rates [30] [28], rapid reconfiguration [29] [16] [14], and high-bandwidth memory access [24] [19] [25]. Rather, we believe that the limited applicability of reconfigurable technology derives largely from the lack of any unifying compute model to abstract away the fixed resource limits of devices which, otherwise, restrict software expressibility as well as longevity across device generations.

A significantly reduced extended abstract of this article appears in FPL'2000 and is copyright Springer-Verlag. This expanded version can be found at <http://www.cs.berkeley.edu/projects/brass/documents/score_tutorial.html>

Existing targets are non-portable. Software for reconfigurable hardware is typically tied to a particular device (or set of devices), with limited source compatibility and no binary compatibility even across a vendor-specific family of devices. Redeploying a program to bigger, next-generation devices, or alternatively to a smaller, cheaper or lower-power device typically requires substantial human effort. At best, it requires a potentially expensive pass through mapping tools. At worst, it requires a significant rewrite to fully exploit new device features and sizes. In contrast, a program written for microprocessor systems can automatically run and benefit from additional resources on any ISA-compatible device, without recompilation.

Existing targets expose fixed resource limitations. The exposure of fixed resource limitations in existing programming models tends to impair their expressiveness and broad applicability. In such programming models, an application's choice of algorithm and spatial structure is restricted by the size of available hardware. Furthermore, a computation's structure and size must be fixed at compile time, with no allowance for dynamic resource allocation. Hence algorithms with data-dependent structures or potentially unbounded resource usage cannot be easily mapped to reconfigurable hardware.[1]

Virtualize resources The SCORE compute model introduced in this paper addresses the issue of fixed resource limits by virtualizing the computational, communication, and memory resources of reconfigurable hardware. FPGA configurations are partitioned into fixed-size, communicating pages which, in analogy to virtual memory, are "paged-in" or loaded into hardware on demand. Streaming communication between pages which are not simultaneously in hardware may be transparently buffered through memory. This scheme allows a partitioned program to run on arbitrarily-many physical pages and to automatically exploit more available physical pages without recompilation. With proper hardware design, this scheme permits binary compatibility and scalability across an architectural family of page-compatible devices.

Convenient and Efficient Model For software to benefit from additional physical resources (pages), the programming model should expose (page-level) parallelism and permit spatial scaling. SCORE's programming model is a natural abstraction of the communication which occurs between spatial, hardware blocks. That is, the data flow communication graph captures the blocks of computation (operators) and the communication (streams) between them. Once captured, we can exploit a wealth of well-known techniques for efficiently mapping these computational graphs to arbitrary-sized hardware. Furthermore, run-time composition of graphs is supported, enabling data-driven program structure, dynamic resource allocation, and the integration of separately compiled or developed library components.

Section 2 of this paper discusses other systems and compute models which have influenced the formulation of SCORE. Section 3 presents the key components of the SCORE model. Section 4 discusses the hardware requirements for a SCORE implementation and why they are reasonable in today's technology. Section 5 gives a brief introduction to programming constructs for SCORE. Section 6 show an execution sample, and Section 7 describes the basic architecture for our current implementation of the SCORE run-time system. Section 8 shows results from a JPEG encoder in SCORE as an example of our early experience implementing a SCORE system.

[1]Data-dependent computational structures can be constructed via specialization and recompilation, as in [38], but this requires a complete pass through mapping tools.

2 Related Work

The technique of time-multiplexing a large spatial design onto a small reconfigurable system was demonstrated by Villasenor *et al.* [31]. By hand-partitioning a particular design (motion-wavelet video coder) into a graph of FPGA-sized "pages" and manually reconfiguring each device with those pages, they were able to run the design on one third as many devices (*i.e.* physical pages) as were originally required with only 10% performance overhead. The key to this approach's efficiency was to amortize the cost of reconfiguration by having each page process a sizable stream of data (buffered through memory) before reconfiguring. SCORE aims to automate the partitioning and efficient dynamic reconfiguration performed manually by Villasenor.

The ease and success of such automation depends on appropriate models for program description and dynamic reconfiguration. In this regard, SCORE builds on prior art developing ISA, data flow, distributed, and streaming computation models. In the remainder of this section, we discuss the relation of SCORE to these prior models.

ISA Models

The first attempts to define a "compute model" for reconfigurable computing devices were focussed on augmenting a traditional processor ISA with *reconfigurable instructions*. PRISC [27] (and later Chimæra [15]) allowed the definition of single-cycle, Programmable Function Unit (PFU) operations using a TLB-like management and replacement scheme to *virtualize* the space of PFU instructions, exploiting local, dynamic reuse of PFU instructions. The size of the PFUOP, itself, however, was fixed by the architecture and PFUOPs are constrained by the sequential ISA to execute sequentially. Hence, the model does not directly, allow the architecture to scale and exploit additional parallel hardware.

DISC [36] and GARP [16] expand the PRISC model to allow variable-size and multiple-cycle array configurations. These architectures can pack multiple configurations (instructions) into the available array and, in the case of GARP, support an implementation dependent number of cached array configurations. However, like PRISC, each array configuration must be smaller than the available, physical logic, and reconfigurable instructions can only be composed sequentially in the ISA. Consequently, these architectures also prevent one from scaling array size and automatically exploiting the additional parallel hardware.

OneChip [19] expands the ISA extension model further by allowing scoreboarded operations from memory to memory in the ISA. While still based on a sequential ISA computing model, this potentially facilitates the use of multiple, parallel RFUs. As long as each RFUOP operates on independent memory banks, the RFU operations will not interlock and may proceed in parallel. This technique, however, exposes the memory buffers between pipelined and chainable operations. It forces the user or compiler to pick blocking factors and to schedule blocked operations in parallel in the processor's instruction dispatch window. In fact, this approach prevents direct pipeline assembly of chained operations. Furthermore, the ISA forces the compiler to schedule the invocation of RFU operations, limiting the opportunity to schedule components of the reconfigurable computation in a data-driven manner.

Dynamic Reconfiguration

Ling and Amano [22] describe the Multi-Processor WASMII, a scalable FPGA-based architecture which partitions and time-multiplexes large applications as FPGA-sized pages, much like SCORE. The primary limitation to WASMII's performance is that page communication is buffered through a small, *fixed* set of device registers (the "token router"). With such a small communication buffer, a page can operate for only a short time before depleting available inputs or output space and, if the page is time-multiplexed, triggering reconfiguration. Hence when running a design which is larger than available hardware, execution time may be dominated by reconfiguration time. Brebner [5] [6] proposes a similar demand-paged, reconfigurable system based on arbitrary-sized *swappable logic units* (SLUs) which communicate through periphery registers[2] and are subject to the same inefficiency as WASMII when time-multiplexed. SCORE avoids this inefficiency by allowing large (unbounded) communication buffers, enabling longer page execution between reconfigurations.

CMU's PipeRench [14] defines a reconfigurable fabric paged into horizontal *stripes* which communicate vertically as a pipeline. The execution model fully virtualizes stripes and enables hardware scaling to any number of physical stripes. Although stripes communicate through input-output registers as in WASMII, PipeRench's stripe-sequential, pipelined reconfiguration scheme hides the excessive reconfiguration overhead seen in WASMII. This sequential reconfiguration scheme is well suited to simple, feed-forward pipelines. However, this scheme does not support computation graphs with feedback loops, and it may waste available parallelism when squeezing wide graphs into a linear sequence of stripes. In particular, when virtualizing a computation with more parallelism than is available in a single architected stripe, non-communicating stripes which simultaneously fit into hardware must still load in sequence, incurring added latency and an area cost for buffering stripe I/O. SCORE makes no such restrictions on execution order, allowing parallel reconfiguration of physical pages.

[2]In Brebner's "parallel harness" model, SLUs are arranged in a mesh and communicate with nearest neighbors via periphery registers. In the data-parallel "sea of accelerators" model, SLUs do not communicate with each other and so would not incur the same virtualization overhead discussed above.

Data Flow

The original Dennis formulation of data flow [11] [10] described a processor ISA which represented data flow graphs directly, each instruction being an operator. The execution model included only a single result register per instruction, allowing an instruction to execute only once at a time before its successors must execute. While this restriction on instruction ordering is reasonable for a microprocessor where large instruction store and fast instruction issue are available, it is not reasonable for a reconfigurable device where reconfiguring on each instruction issue is too costly. Iannucci's *hybrid data flow* [18] and Berkeley's TAM [9] define operators by straight-line blocks of instructions, relaxing the frequency of inter-instruction synchronization to only the entry and exit points of blocks. Nevertheless, these models inherit the same problem of fixed communication buffers as Dennis data flow and thus face the same inefficiency as WASMII in a time-multiplexed reconfigurable implementation.

Streaming formulations of data flow remove the limitation of fixed input-output buffers, allowing arbitrarily many tokens to queue up along an arc of a data flow graph. This generalization allows a time-multiplexed implementation to fire an operator many times in succession before reconfiguring, amortizing the cost of reconfiguration over a large data set. Lee's synchronous data flow (SDF) [21] [3] incorporates streaming for the restricted case of static flow rates. Although this model of computation is not Turing-complete (it lacks data-dependent control flow), it guarantees that conforming graphs can be statically scheduled to run with bounded stream buffers.

Buck's integer-controlled data flow (IDF) [7] incorporates data-dependent control flow by adding to SDF a set of canonical dynamic-rate operators (*e.g. switch*, *select*). SCORE permits a dynamic-rate model, allowing data-dependent control flow inside any operator. As such, SCORE programs are essentially equivalent to IDF in expressiveness, since a SCORE operator is equivalent to an IDF graph containing dynamic operators.

SCORE shares a gross similarity to heterogeneous systems which use streaming data flow to tie together arbitrary processors (conventional, special-purpose, and/or reconfigurable) including MIT's Cheops [4], MagicEight [35] [34], and Berkeley's Pleiades [26] [1]. The programming model of these systems is more restricted than SCORE, typically based on a pre-defined set of streaming operations. Furthermore, SCORE provides a stronger abstract model allowing pages (processors) to be swapped as needed and hiding implementation limitations like buffer sizes.

Streaming APIs

Virginia Tech [20] defines a streaming API for processor-controlled networks of reconfigurable devices (*e.g.* SLAAC [8]). While standardizing the form in which applications may be written, this API does not, in and of itself, virtualize the size or fabric of compute resources and hence does not allow the definition of portable and scalable designs. Rather, it serves only as a hardware interface layer to manually reconfigure devices on the network.

Maya Gokhale [13] defines a C-based streaming programming model. Like SCORE, Streams-C exploits that fact that reconfigurable hardware is efficiently organized as a collection of spatial pipelines and that streams provide a natural abstraction for the hardware linkage between two separate design components. Nevertheless, Streams-C only serves as a convenient way to compactly describe spatial designs. No virtualization is performed, and the burden of handling placement and fixed buffer size restrictions is placed entirely on the programmer. In these regards, SCORE attempts to provide a much higher level programming model, providing semantics which are decoupled from hardware artifacts, like buffer sizes and physical hardware size, and automatically filling in these lower level details at compile and run time.

CSP

In many ways the SCORE computational model is similar to Hoare's Communicating Sequential Processes (CSP) [17]. Each SCORE operator can be viewed as a single process. These operators communicate with each other via designated stream connections somewhat like CSP's named ports. Unlike CSP ports, SCORE streams are buffered and offer an unbounded stream abstraction. Significantly, SCORE operators, unlike CSP processes, are not allowed to be non-deterministic. Composition of SCORE operators always yields deterministic, observable results. In fairness, most of CSP's non-determinism is to facilitate modeling of unpredictable, dynamic effects in real systems, and most of SCORE could be modeled on top of CSP. SCORE also allows dynamic construction of computational graphs, which was not in the original CSP formulation, but could of course be added.

3 SCORE Computational Models

A compute model defines the computational semantics that a developer expects the physical machine to provide. The compute model itself is abstract but captures the essence of how computation proceeds, defining the meaning of any computation. The compute model is given a more concrete embodiment in one or more *programming models*. The programming model provides a high-level view of application composition and execution, adding a number of practical conveniences for the programmer. Ultimately, both models are grounded in an *execution model* which defines the way the computation is actually described to the physical hardware and the meaning associated with any such description.

The execution model, programming model, and abstract computational model are all consistent views of computation. What differs among them is the level of detail which they expose or abstract (See Figure 1 and 2). The execution model abstracts the number of key resources (*e.g.* ALUs, pages) to allow scaling across different hardware platforms. The programming model abstracts architectural characteristics found in the execution model (*e.g.* ISA details, limited resource sizes exposed at architectural level). The compute model abstracts away the concrete syntax and primitives provided by a particular programming language or system.

3.1 Compute Model

A SCORE computation is a graph of computation nodes (operators) and memory blocks linked together by streams. 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 (FSM) nodes which interact with the rest of the graph *only* through their stream links; and (2) Turing complete (TM) nodes which support resource allocation in addition to stream operations.

SCORE FSMs 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 FSM 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, SCORE FSMs transition to a new state based on their inputs and present state. Each SCORE FSM has a distinguished *done* state into which it may enter to signal its completion and to remove itself from the running computation.

A SCORE TM node is similar to a SCORE FSM node but adds the ability to allocate memory and to create new graph nodes (FSM or TM operators) and edges (streams) in the SCORE compute graph.

Memory is allocated in finite-sized blocks called *segments*. Each segment may be owned by a single operator at a time. A SCORE TM may allocate new segments and pass them on to an FSM or TM node that it creates. Upon termination, when a TM or FSM node enters the *done* state,

Compute Model	**Programming Model**	**Execution Model**
Abstract model capturing essential semantics of computation	Particular set of programming constructs providing a convenient way to express computations in the compute model	Low-level (executable) description of the computation and the semantics which the hardware is expected to provide when interpreting this description
	The programming model is abstracted from certain details that arise in the execution model, like architectural page size or number of registers.	The execution model is abstracted from certain hardware size details like number of resources.
sequential execution + single global memory	C+Unix	MIPS-ISA + Unix-ABI
	C+WinAPI	x86-ISA + WinABI
SCORE	C++ + TDF + ScoreRT + linux	MIPS-ISA + linux-ABI + SCORE 256 4-LUT CPs + SCORE <1MB segments
CSP	Occam	Transputer ISA
TTDF	Id	TL0-sparc + AM + Solaris
SPMD	C*	Sparc-ISA + CM5-runtime + Solaris
vector	vectorized C	T0-ISA SunOS-ABI
SDF	Ptolemy graphs	TMS320C40

Figure 1: Levels of Abstraction for Computational Model

Computational Element	**Compute Model**	**Programming Model**	**Execution Model**	**Hardware**
Compute	SCORE FSM	Operator	Page	Physical Compute Page (CP)
Communication	Stream	Stream	Stream	• physical network • CMB • main memory
Data Storage	Segment	Segment	Segment	• CMB • main memory

Figure 2: SCORE Computational Elements at Various Levels of Abstraction

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.

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-accessor, 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.

Appendix B defines the compute model more precisely.

3.2 Programming Model

A programming model gives the programmer a framework for describing a computation in a manner independent of device limits, along with guidelines for efficient execution on any hardware implementation. It can be more abstract than the execution model because the compiler will take care of translating the higher level description provided by the programmer into the details needed for execution. The key abstractions of the SCORE programming model are *operators*, *streams*, and *memory segments*.

3.2.1 Basic Components

Operators

An *operator* represents a particular algorithmic transformation of input data to produce output data. Operators are the computational building blocks for a computation (*e.g.* multiplier, FIR filter, FFT). Operators may be behavioral primitives or hierarchical graph compositions of other operators. Figure 3 shows an example video processing operator composed as a pipeline of transformations, including a *motion estimation* operator, an image *transformation* operator, a data *quantization* operator, and a *coding* operator. The size of an operator in hardware is implementation dependent and is in no way limited in the programming model. Operators may need to be partitioned to fit onto an architectural compute page. Partitioning is an integral part of the automated in the compilation process.

Streams

Inter-operator communication uses a streaming data flow discipline. When the programmer needs to connect operators together, he links the producer to the consumer operator using a *stream* link. The link both serves to define where data is logically routed and acts as an unbounded-length queue for data tokens. Operators signal both when they are producing data and when they need to consume data. This signalling translates into data presence signals on the stream links which synchronize all communication between operators.

Memory Segments

A memory segment is a contiguous block of memory and serves as the basic unit for memory management. Memory segments may be any size, up to an architecturally defined maximum. A memory segment may be used in a SCORE computation by giving it a specific operating mode (*e.g.* sequential read, random-access read-write, FIFO) with appropriate stream interface, then linking it into a data flow graph like any other operator (see Figure 5).

3.2.2 Dynamic Features

On top of these basic components, SCORE supports a number of important dynamic features.

- Dynamic rate operators
- Dynamic graph composition and instantiation
- Dynamic handling of uncommon events

Dynamic rate operators

An operator may consume and produce tokens at data-dependent rates. This expressive power allows SCORE to describe efficient operators for tasks such as data compression, decompression, and searching or filtering. Section 5.2 shows a possible set of linguistic constructs for supporting dynamic rate consumption and production. To exploit dynamic rates, scheduling decisions should be made at run time, when the dynamic rates and actual data availability are known.

Dynamic composition and instantiation

SCORE allows run-time instantiation of operators and data flow graphs. That is, the computational graph may be created, extended, or modified during execution. Extending the graph means creating new graph nodes and edges which may be defined in a data-dependent manner. An operating node may terminate during execution, and existing stream links may be shut down by their attached operators.

This mechanism has several benefits over describing a computation strictly by a static graph at compile-time. It gives the programmer an opportunity to postpone or avoid allocating resources for parts of the computation which are not used immediately or whose resource requirements cannot be bound until run time. It also enables the creation of data-dependent computational structures, for in-

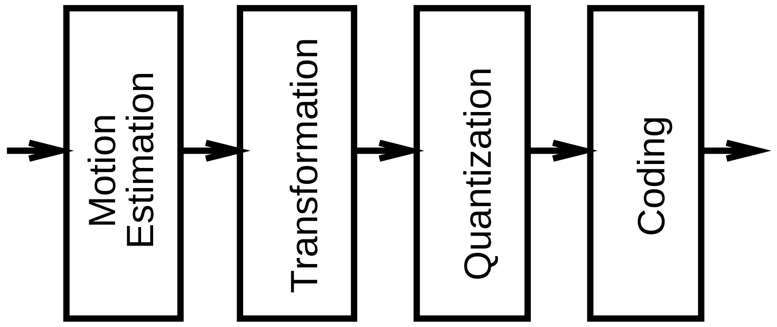


Figure 3: Video Processing Operator

stance, to exploit dynamically-unrolled parallelism. Finally, it creates a framework in which aggressive implementations may dynamically specialize operators around instantiation parameters. That is, an operator may have *parameters* bound at *instantiation time*—*i.e.* when the operator is composed into a data flow graph. This mechanism allows operators to be initialized with unchanging or slowly changing scalar data or to be specialized around parameter values. Examples in Section 5.2 show one set of linguistic constructs to support composition and instantiation.

Exception handling

Exception handling falls naturally out of the data flow discipline of SCORE. When an unusual condition occurs, the operator may raise an exception. At this point, the operator stops rather than producing output data. Dependent, downstream operators may have to stall waiting for this operator to resume and produce an output, but the data flow disciplines guarantees that they wait properly for the operator to handle the exception and produce a result. When the exception is handled, the raising operator resumes operation, producing data, and allowing the downstream operators to resume in turn.

3.3 Execution Model

The key idea of a computer architecture is that it defines the computational description that a machine will run and the semantics for running it (*e.g.* the x86 ISA is a popular architectural definition for processors). Someone building a conforming device is then free to implement any detailed computer organization that reads and executes this computational description (*e.g.* i80286, i80386, i80486, Pentium, and K6 are all different implementations that run the same x86 computational description). Following this technique, the execution model for SCORE defines the run-time computational description for an architecture family and the semantics for executing this description.

The SCORE execution model defines all computation in terms of three key components:

- A *compute page* (CP) is a fixed-size block of reconfigurable logic which is the basic unit of virtualization and scheduling.
- A *memory segment* is a contiguous block of memory which is the basic unit for data page management.
- A *Stream link* is a logical connection between the output of one page (CP, segment, processor, or I/O) and the input of another page. Stream implementations will be physically bounded, but the execution model provides a logically unbounded stream abstraction.

A computational description in this execution model is independent of the size of the reconfigurable array, admitting architectural implementations with anywhere from one to a large number of compute pages and memories. The model provides the semantics of an unlimited number of independently operating physical compute pages and memory segments. Compute pages and segments operate on stream data tagged with input presence and produce output data to streams in a similar manner. The use of data presence tags provides an operational semantics that is independent of the timing of any particular SCORE-compatible computing platform.

Fixed Compute-Page Sizes

Compute pages are the basic unit of virtualization, scheduling, reconfiguration, and relocation. In analogy with a virtual memory page, a compute page is the minimum unit of hardware which is mapped onto physical hardware and is managed as an atomic entity. Each compute page represents a fixed-size piece of reconfigurable hardware (*e.g.* 64 4-LUTs). Compute pages differ from the operators of the compute model in that pages have architecturally imposed resource limitations such as size and maximum number of streams.

The decomposition of a computation into compute pages takes the stand that it is not feasible nor desirable to manage every primitive computational building block (*e.g.*

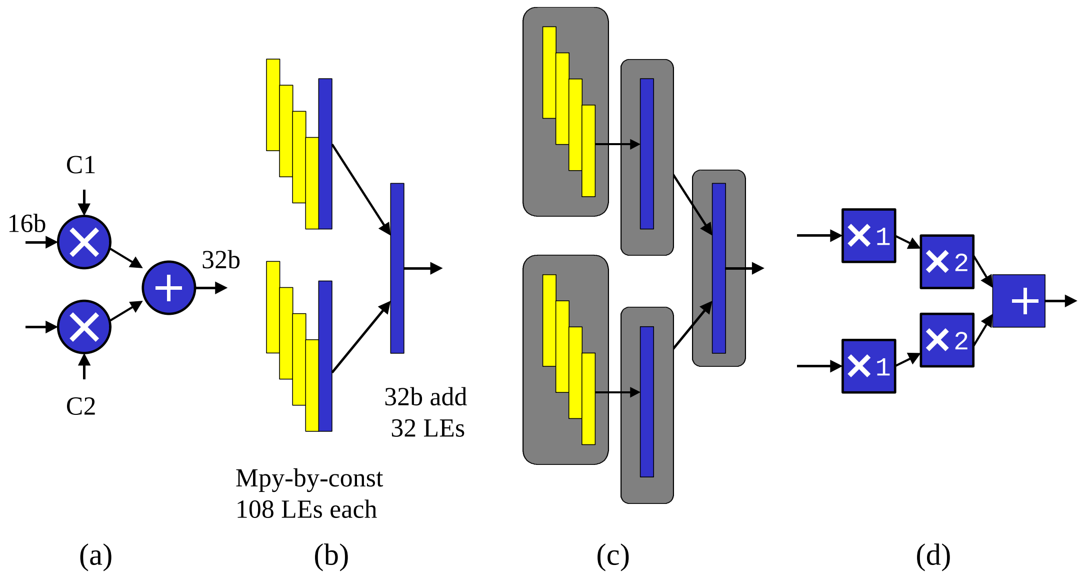


Figure 4: Example of Page Decomposition: (a) original operator as seen in programming model, (b) mapped to logic elements (LEs) in target architecture, (c) partitioning into 64-LE pages to match execution model page size, (d) final graph used by execution model

4-LUT) as an independent entity—just as it is generally not desirable to manage every bit of memory as an independent block. Rather, by grouping together a larger block of resources, management and overhead can be amortized over the larger number of computational blocks. This grouping also allows hard problems, like placement and routing, to be performed offline within each page. Note that it is necessary that the page size be fixed across an architecture family so that all family member can run from the same run-time (binary) description. Otherwise, page (re-)packing, placement, and routing would need to be performed online. The fixed page discipline requires that compilers partition (or pack) more abstract computational operators into these fixed size pages. Figure 4 shows an example decomposition of an operator graph into pages.

Compute pages may contain internal state which must be saved and restored when the page is swapped onto or off of a physical compute page. Swapping may be necessary in a time-multiplexed implementation and is key to supporting the semantics of an unbounded number of compute pages.

Memory Segments and Configurable Memory Blocks A memory segment is a contiguous block of memory which is managed as a single, atomic memory block for the purposes of swapping and relocation. A memory segment may be used in one of several modes (*e.g.* sequential read, random-access read-write, FIFO). When configured

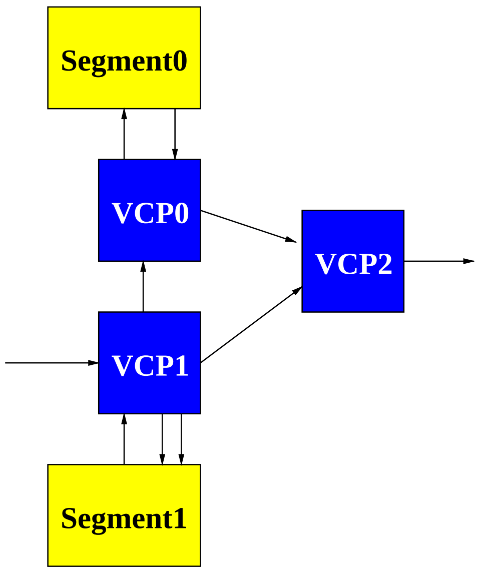


Figure 5: Data Flow Computation Graph with both Compute Pages and Segments

into a particular mode, a segment has logical stream ports to connect it to the graph of pages (*e.g.* streams for random access: address input, data input, data output, control input). Figure 5 shows an example graph connecting pages and segments.

To use a memory segment, the run-time system will map it into a *configurable memory block* (CMB). The CMB is a physical memory block inside the reconfigurable array (See, for example, Figure 11) with active stream links and interconnect to connect the memory segment into the active computation. In addition to holding user-specified segments, CMBs are also used to hold segments containing CP configurations, segments containing CP state, and segments associated with stream buffers. A single CMB may hold any number of each of these types of segments as long as their aggregate memory requirement does not exceed the CMB's capacity (see Figure 6 for a sample memory layout in a CMB). In our current vision, only a single such segment may actually be active in each CMB at any point in time, but there is nothing in the SCORE definition that prevents an implementation from being designed to handle multiple, active segments in the same CMB.

Physically Finite, Logically Unbounded Streams

Streams form the data flow links between pages. A page (CP or segment) indicates when it is producing a valid data output with an out-of-band *data present* bit. The valid data value with its associated presence bit is termed a *token*. The token is transported to the destination input of the consuming operator. The stream delivers all data items generated by the producer, in order, to the consumer, storing each until the consumer indicates it has consumed it from the head of its input queue (See Figure 7). The data presence tag in a token serves a similar role to a *stall* signal in a conventional virtual memory or cache architecture; that is, it lets the processing unit know if data is available and it can continue processing or if the processing unit must wait for data to arrive.

When a stream is empty, the downstream operator will stall waiting for more input data. This discipline hides the detailed timing of operations from the programming model, guaranteeing correct behavior while allowing variations between implementations of the computing architecture.

Even at the run-time level, streams provide the abstraction of unbounded capacity links between producers and consumers.[3] In practice, however, the streams are finite, with an implementation-dependent buffer capacity. To implement the semantics of unbounded, FIFO stream links, an implementation will use *backpressure* (See Figure 7) to stall production of data items, and the run-time system will allocate additional buffer space in FIFO segments as needed (See Figure 8 for an example of stream buffer expansion).

Physically, a virtual stream may be realized in one of two ways:

- When both the producer and the consumer of a virtual stream are loaded on the physical hardware, the stream link can be implemented as a spatial connection through the inter-page routing network between the two pages.[4] (See Figure 9.)
- When one of the ends of the stream is not resident, the stream data can be sinked (or sourced) from a stream buffer segment active in some CMB on the component. (See Figure 10.)

This allows efficient, pipelined chaining of connected operators when space permits, as well as deep, intermediate data buffering when a computation must be sequentialized.

Hardware Virtualization

Compute pages, segments, and streams are the fundamental units for allocation, virtualization, and management of the hardware resources. At run time, an operating system manager schedules virtual pages and streams onto the available physical resources, including page assignment and migration and inter-page routing.

If there are enough physical resources, every page of a computation graph may be simultaneously loaded on the reconfigurable hardware, enabling maximum-speed, *fully-spatial* computation. Figure 9 shows this case for the video processing operator of Figure 3. If hardware resources are limited, a computation graph will be time-multiplexed onto the hardware. Streams between virtual pages that are not simultaneously loaded will be transparently buffered through on-chip memory. Figure 10 shows this case for the video processing operator. Each component operator is loaded into hardware in sequence, taking its input from one memory buffer and producing its output to another.

3.4 Model Implications

3.4.1 Advice for Programmers

One goal of the compute model is, at a high-level, to focus the developer on the style of computation which is efficient for the hardware and execution model. To better utilize scalable reconfigurable hardware, SCORE developers should:

- *Describe computations as spatial pipelines with multiple, independent computational paths.* A hardware

[3]See Appendix A for a discussion of why unbounded buffers are necessary.

[4]An implementation could choose to implement this link as a statically configured path as in FPGAs, a time-switched path, or even a dynamically routed path.

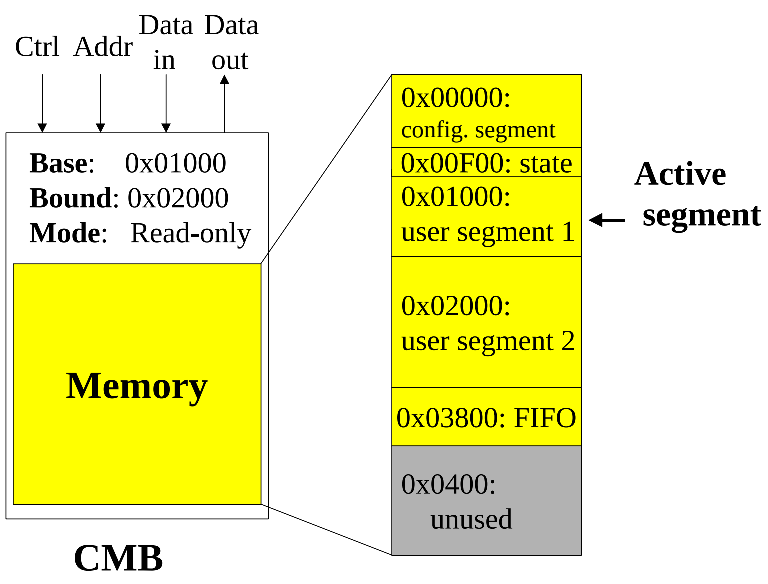


Figure 6: Segments and Other Data mapped onto a CMB

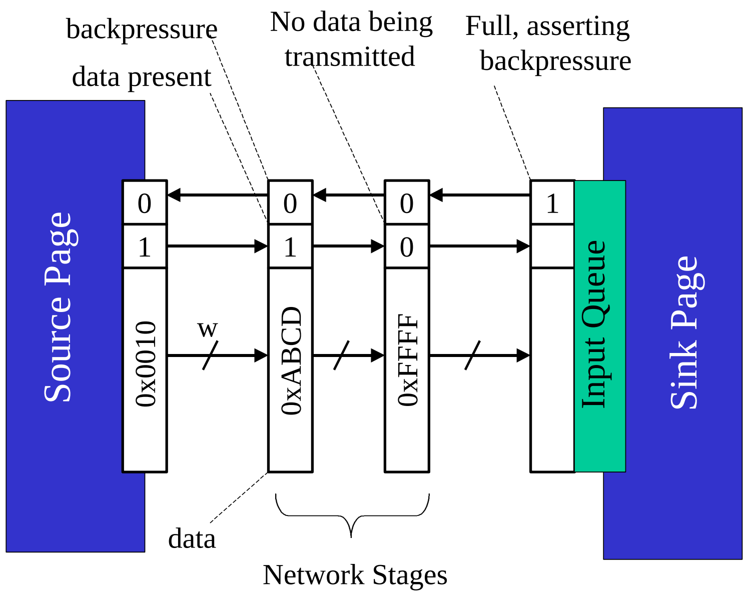


Figure 7: Stream Signals

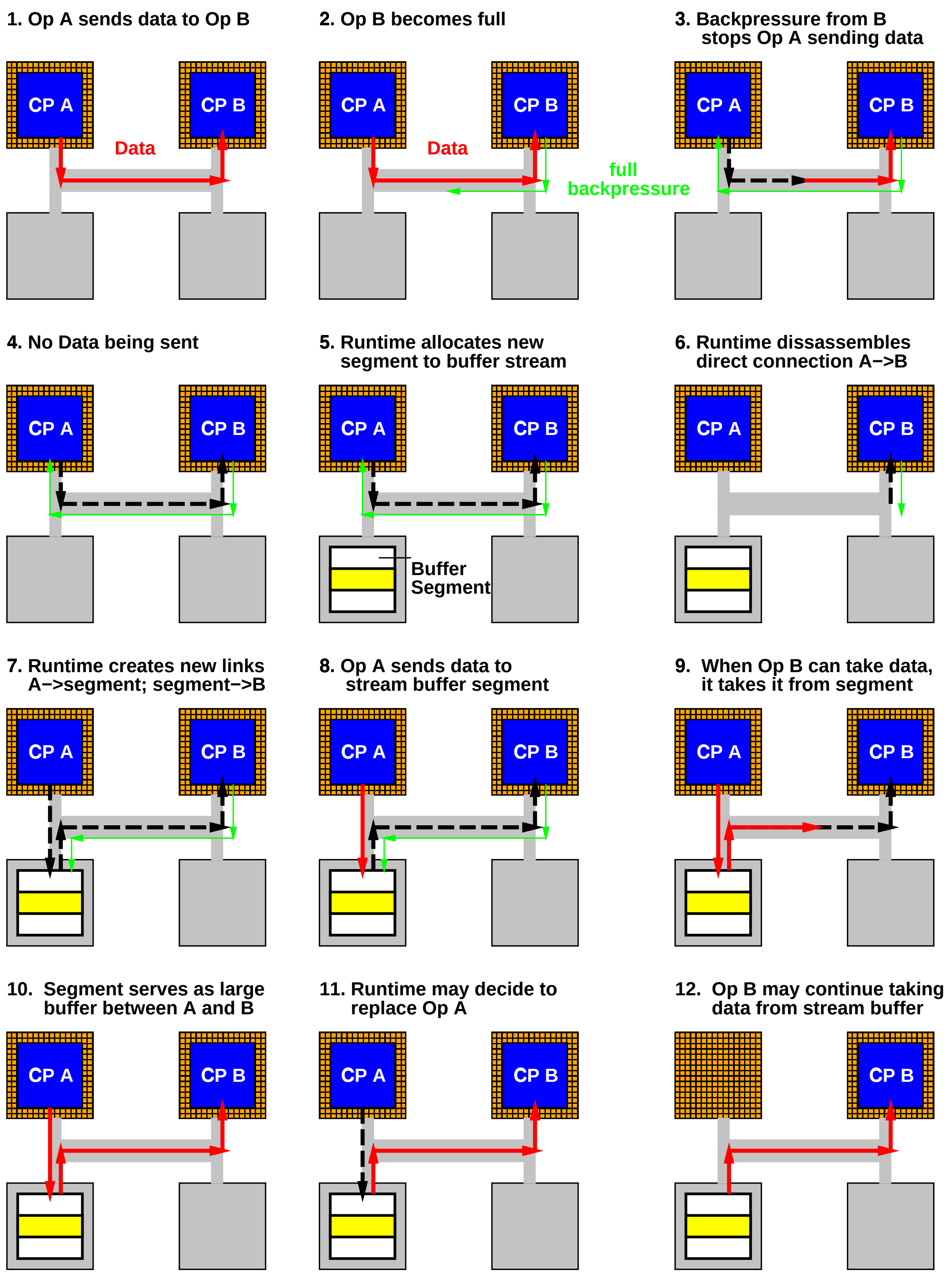

1. Op A sends data to Op B
CP A
CP B
Data
2. Op B becomes full
CP A
CP B
Data
full
backpressure
3. Backpressure from B stops Op A sending data
CP A
CP B
4. No Data being sent
CP A
CP B
5. Runtime allocates new segment to buffer stream
CP A
CP B
Buffer Segment
6. Runtime dissassembles direct connection A−>B
CP A
CP B
7. Runtime creates new links A−>segment; segment−>B
CP A
CP B
8. Op A sends data to stream buffer segment
CP A
CP B
9. When Op B can take data, it takes it from segment
CP A
CP B
10. Segment serves as large buffer between A and B
CP A
CP B
11. Runtime may decide to replace Op A
CP A
CP B
12. Op B may continue taking data from stream buffer
CP B

Figure 8: Expansion of Finite Stream Buffer to provide Unbounded Stream Buffer Semantics

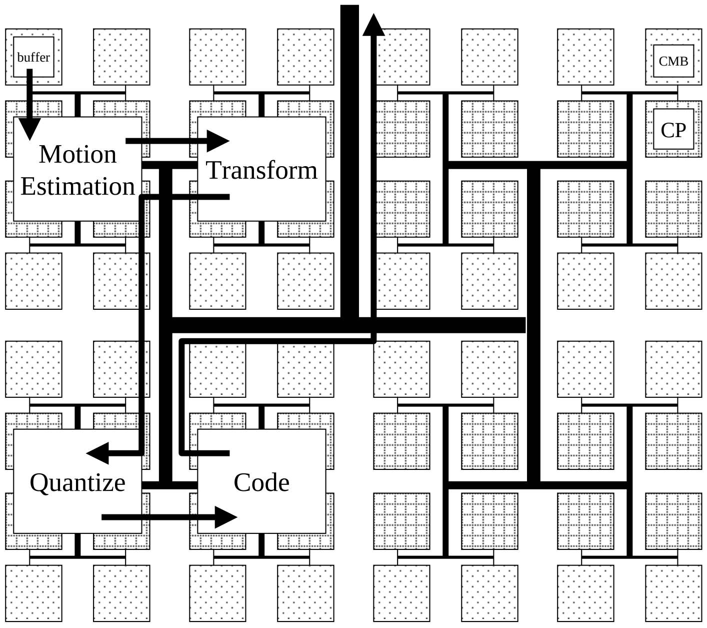


Figure 9: Fully Spatial Implementation of Video Processing Operator on Abstract SCORE Hardware

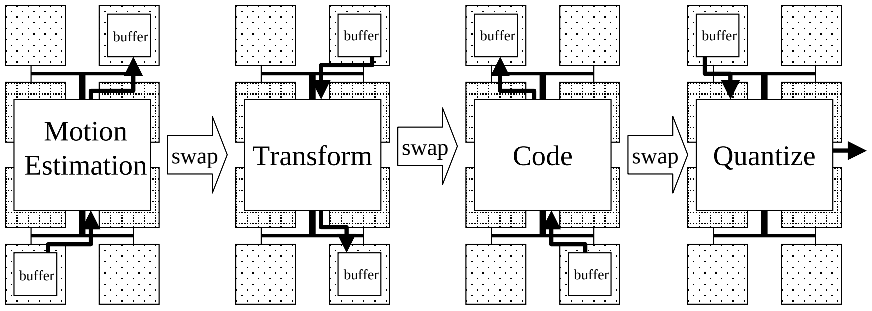


Figure 10: Capacity-Limited, Temporal Implementation of Video Processing Operator

implementation will attempt to concurrently execute as many of the specified, parallel paths as possible.

- *Avoid or minimize feedback cycles.* Cyclic dependencies introduce delays which cannot be pipelined away and hence increase the total run time or lead to page-thrashing in small hardware implementations.

- *Expose large data streams to SCORE operators.* Large data sets help amortize the overhead of loading computation into reconfigurable hardware, especially into small, time-multiplexed hardware implementations.

3.4.2 Generality

While we have described the SCORE hardware model here in terms of a single processor and homogeneous computational pages and memories, the model itself admits a number of extensions. SCORE can accomodate heterogeneous and specialized computational pages, as seen in Pleiades [26] and Cheops [4]. Using specialized pages most efficiently makes the scheduling problem more interesting, since some operators may run on multiple kinds of specialized pages. Also, there is nothing which prohibits SCORE from using multiple conventional processors for executing sequential operators and/or the run-time scheduler. Conventional techniques for multiprocessing and distributed scheduling would be relevant in this case.

4 Hardware Requirements

SCORE assumes a combination of a sequential processor and a reconfigurable device. The reconfigurable array must be divided into a number of equivalent and independent compute pages.[5] Multiple, distributed memory blocks are required to store intermediate data, page state, and page configurations.

The interconnect among pages is critical to achieving high performance and supporting run-time page placement. It should support high bandwidth, low latency communication among compute pages and memory, allowing memory pages to be used concurrently. The interconnect must buffer and pipeline data as well as provide back-pressure signals to stall upstream computation when network buffer capacity is exceeded. Routing resources should be sufficiently rich to facilitate rapid, online routing.

The compute pages themselves may use any reconfigurable fabric that supports rapid reconfiguration, with provision to save and restore array state quickly. The BRASS HSRA subarray design [30] is a feasible, concrete implementation for a compute page. It provides microsecond reconfiguration and high-speed, pipelined computation.

Each *configurable memory block* (CMB) is a self-contained unit with its own stream-based memory port and an address generator (see Figure 6). CMBs may be accessed independently and concurrently in a scalable system. The memory fabric may use external RAM or on-chip memory banks (*e.g.* BRASS Embedded DRAM [25]), with additional logic to tie into the data flow synchronization used by the interconnect network. The memory controllers need to support a simple, paged segment model including address relocation within a memory block and segment bounds. Streaming data support obviates the need for external addressing during reconfiguration and stream buffering.

The sequential processor plays an important part in the SCORE system. It runs the page scheduler needed to virtualize computation on the array, and it executes SCORE operators that would not run efficiently in reconfigurable implementation. Consequently, the processor must be able to control and communicate with the array efficiently. A single-chip SCORE system (*e.g.* see Figure 11) integrating a processor, reconfigurable fabric, and memory blocks could provide tight, efficient coupling of components.

Although a single-chip SCORE implementation offers benefits for performance and design efficiency, the SCORE model permits a wide range of implementations including one using conventional, commercial components.

[5]In a degenerate case, there can be only one page, but this sacrifices many of the strengths of the SCORE model.

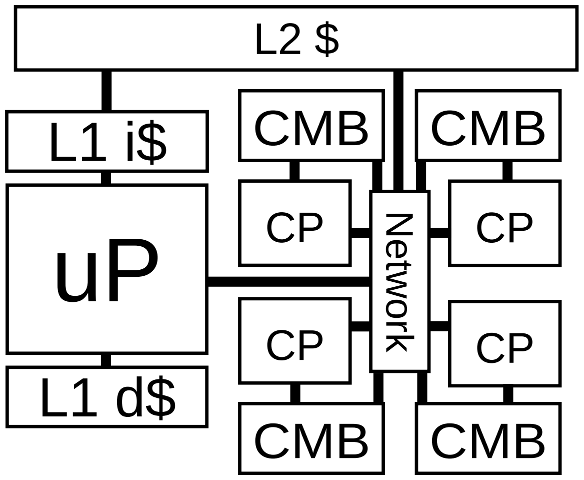


Figure 11: Hypothetical, single-chip SCORE system

5 Language Instantiations

As a computational model, any number of languages which obey the SCORE semantics could be defined to describe SCORE computations. One could define subsets of conventional HDLs (*e.g.* Verilog, VHDL) with stylized Input/Output primitives to describe SCORE operators and operator composition. Similarly, one could define subsets of conventional programming languages (*e.g.* C++, Java) to perform these tasks. To focus on the necessary semantics, we have defined an intermediate register-transfer level language (RTL) to describe SCORE operators and their composition for our initial development work. We view our intermediate language, TDF, as a device-independent, assembly language target on the way to architecture-specific executable operators.

5.1 SCORE Language Requirements

As indicated by the semantics of the SCORE compute model, SCORE operators are synchronous, single clock entities, with their own state. Operators communicate *only* through designated I/O streams. Operation is gated by data presence on the I/O streams. As such, each operator can be viewed as a finite-state machine with associated data path (*i.e.* FSMD [12]). In a multithreaded language, such as Java or C++ with an appropriate thread package, a SCORE operator would be an independent thread which communicates with the rest of the program only through single-reader, single-writer I/O streams. Specifically, SCORE does not have a global, shared-memory abstraction among operators. An operator may *own* a chunk of the address space (a memory segment) during operation and return it after it has completed, but no two operators may simulta-

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

Figure 12: TDF Specification of 4-TAP FIR (a static rate operator)

neously own a piece of memory.

5.2 TDF

TDF is basically an RTL description with special syntax for handling input and output data streams from the operator. Common data path operators can be described using a C-like syntax. For example, Figure 12 shows how an FIR computation might be implemented in TDF. Operators may have parameters whose values are bound at operator instantiation time; parameters are identified with the keyword `param`. In the FIR example, the coefficient weights are parameters; these are specified when the operator is created and the values persist as long as the operator is used. The FIR defines a single input stream (`x`) and produces a single output stream (`y`). The behavior of the state is gated on the arrival of the next `x` input value, producing a new `y` output for each such input.

To allow dynamic rate operators, the basic form of a behavioral TDF operator is that of a finite-state machine. Each state specifies the inputs which must be present before it can fire. Once the inputs arrive, the operator consumes the inputs and the FSM may choose to change states based on the data consumed from the inputs. A simple `merge` operator is shown in Figure 13, demonstrating how the state machine can also be used to allow data dependent consumption of input values. Output value production can be conditioned as shown in Figure 14. Together, these allow the user to specify arbitrary, deterministic, dynamic-rate operators.

Of course, the FSM gives the user the semantic power to describe heavily sequential and complex, control-oriented

N.B. *This version has been simplified for illustration; It does not properly handle the end-of-stream condition.*

```
signed[w] merge(param unsigned[6] w,
          // can use parameters to define
          //   data width
                       input signed[w] a,
                       input signed[w] b)
 {
   signed[w] tmpA;
   signed[w] tmpB;
   // states used here to show dynamic
   //   data consumption
   state start(a,b):
    {
      tmpA=a;  tmpB=b;
      if (tmpA<tmpB) { merge=tmpA;
                       goto replaceA; }
      else { merge=tmpB;
             goto replaceB; }
      // note: assignment to function name
      //  signifies output on operator
      //  ''return'' output stream
    }
   state replaceA(a):
    {
      tmpA=a;
      if (tmpA<tmpB) { merge=tmpA;
                       goto replaceA; }
      else { merge=tmpB;goto replaceB; }
    }
   state replaceB(b):
    {
      tmpB=b;
      if (tmpA<tmpB) { merge=tmpA;
                       goto replaceA; }
      else { merge=tmpB; goto replaceB; }
    }
  }
```

Figure 13: TDF Specification of `merge` Operator (a dynamic input rate operator)

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

Figure 14: TDF Specification of `uniq` Operator (a dynamic output rate operator)

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

Figure 15: TDF Compositional Operator

operators. Nonetheless, the programmer should avoid sequentialization and complex control when possible, as operator with many states are less likely to use spatial computing resources efficiently.

Larger operators can be composed from smaller operators in a straight-forward manner as shown in Figure 15.

5.3 C++ Integration and Composition

With a suitable stream implementation and interface code, SCORE operators can be instantiated by and used with a conventional, multithreaded programming language. Figure 16 shows an example C++ program which uses the `merge` and `uniq` operators defined here. Note that SCORE operator instantiation and composition can be performed from the C++ code. Once created, the SCORE operators behave as independently running threads, operating in parallel with the main C++ execution thread. In general, a SCORE operator will run until its input streams are closed or its output streams are freed.

Once primitive behavioral (or leaf) operators are defined (*e.g.* in TDF or some other suitable form) and compiled into their page-level implementation, large programs can be composed entirely in a programming language as shown here. If one thinks of TDF as a portable assembly language for critical computational building blocks, then this language binding allows a high-level language to compose these building blocks in much the same way that assembly language kernels have been composed using high-level languages in order to efficiently program early DSPs and supercomputers. The instantiation parameters for TDF operators allow the definition of generic operators which can be highly customized to the needs of the application.

6 Execution Example

The following example demonstrates execution of the design in Figure 16. It shows array compute page reconfiguration, execution of scheduled behavioral code, and some fundamental control signals.

To ground this explanation to a particular hardware configuration and its constraints, we make the following assumptions about the reconfigurable array parameters and the TDF design in the user application:

- The design consists of three behavioral operators. Full implementation of each operator requires only one compute page.
- The reconfigurable array contains one compute page (CP) and three configurable memory blocks (CMBs).

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

  STREAM_CLOSE(i0); // close input
  STREAM_CLOSE(i1); // streams
  STREAM_CLOSE(i2);
  // output results
  // (for demonstration purposes only)
  for (int cnt=0; !STREAM_EOS(o); cnt++) {
    cout << "result["<< cnt << "]=" <<
       STREAM_READ(o) << endl;
  }
  STREAM_FREE(o);
  return(0);
}
```

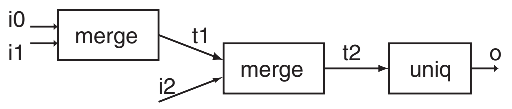


Figure 16: C++ Instantiation and Usage Example

- Each CMB is partitioned into four segments S0 through S3.

  Segments S0 and S1 buffer computation data. In this example, each has a capacity of 15 tokens.

  Segments S2 and S3 store state (FIFO buffers, state machines, and internal registers) and configuration for a compute page.

  CMB state is maintained by its controller, details of which are not shown in this example.

- Each CP has two input and two output FIFO buffers. To make this example clear, the size of the buffers has been set to zero.

- Scheduling and array reconfiguration are performed at the beginning of each timeslice. Refer to the timeline shown in Figure 17 for the execution event sequence.

Table 1 shows the physical view of the array at each point on the timeline in Figure 17. To make diagrams easier to read, single letter identifiers were assigned as follows to each of the operators in the design: A — `merge` with inputs `i0` and `i1`, B — `merge` with inputs `t1` and `i2`, C — `uniq`. The contents of segments S0 and S1 are identified by stream variable name from the program listing in Figure 16 and the first several tokens buffered. The horizontal "empty-full" bar indicates qualitatively the number of tokens present in a segment at a point in time, assuming full segment capacity of 15 tokens.

7 SCORE Run-Time Environment

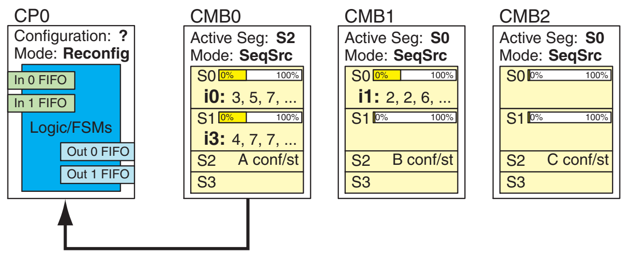
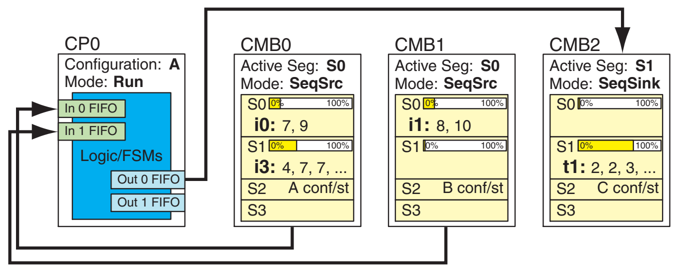
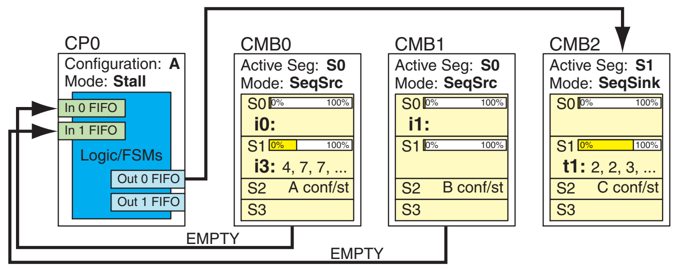
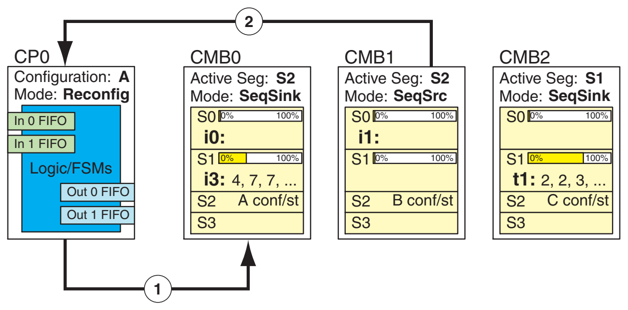
In this section we describe a few of the pragmatics associated with our current run-time architecture and tools for TDF language processing and code generation. These details are not part of the basic SCORE definition, but may help you understand SCORE better by providing a particular, concrete grounding. In particular, this section fills in some of the details between the design shown in Figure 16 and the execution example demonstrated in Table 1.

7.1 Building Applications

Both the SCORE run-time system and user applications are implemented as Linux processes as shown in Figure 19. The compilation and linking process for the user application is shown in Figure 18. The TDF compiler processes the two TDF sources and their corresponding *fuser*[6] files

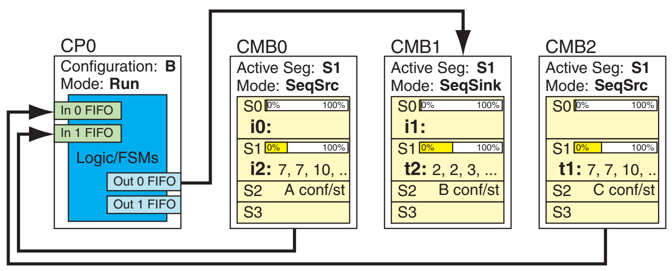
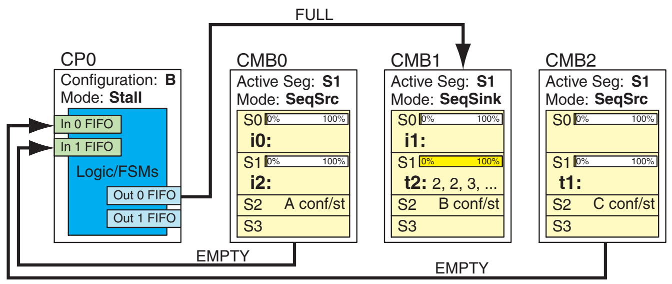
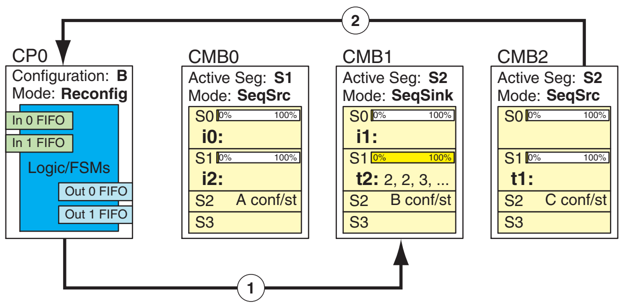
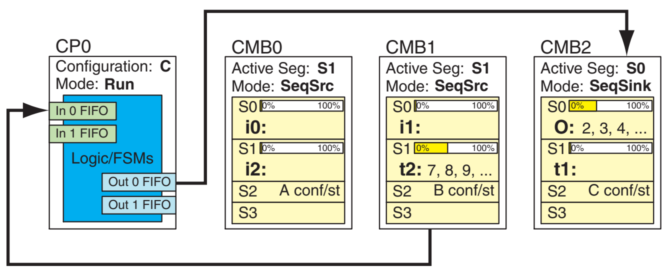
[6]*fuser* files describe parameters of an operator instance. For example, for the operator `uniq(param unsigned[6] w, input signed[w] x)`, the file uniq.fuser contains `uniq(8,)` instructing `tdfc` to produce code for an instance of `uniq` operator which operates on an eight-bit data path.

Table 1: Step-by-Step Execution Example

Time	Physical Array View	Description
I	 CP0 Configuration: ? Mode: Reconfig In 0 FIFO In 1 FIFO Logic/FSMs Out 0 FIFO Out 1 FIFO CMB0 Active Seg: S2 Mode: SeqSrc S0 0% 100% i0: 3, 5, 7, ... S1 0% 100% i3: 4, 7, 7, ... S2 A conf/st S3 CMB1 Active Seg: S0 Mode: SeqSrc S0 0% 100% i1: 2, 2, 6, ... S1 0% 100% S2 B conf/st S3 CMB2 Active Seg: S0 Mode: SeqSrc S0 0% 100% S1 0% 100% S2 C conf/st S3	**Initially assume** that the contents of streams `i0`, `i1`, and `i2` have been loaded by the main processor into segments CMB0 S0, CMB0 S1, CMB1 S0. In addition, the configuration and initial state of pages (operators) A, B, and C has been loaded into segments CMB0 S2, CMB1 S2, and CMB2 S2 respectively. **Reconfiguration.** Page A (`merge`) is scheduled to run for the first timeslice. First, CP0 is configured with the contents of CMB0 S2. Then, the streams are setup between CMBs and the CP0 as shown on the next diagram.
II	 CP0 Configuration: A Mode: Run In 0 FIFO In 1 FIFO Logic/FSMs Out 0 FIFO Out 1 FIFO CMB0 Active Seg: S0 Mode: SeqSrc S0 0% 100% i0: 7, 9 S1 0% 100% i3: 4, 7, 7, ... S2 A conf/st S3 CMB1 Active Seg: S0 Mode: SeqSrc S0 0% 100% i1: 8, 10 S1 0% 100% S2 B conf/st S3 CMB2 Active Seg: S1 Mode: SeqSink S0 0% 100% S1 0% 100% t1: 2, 2, 3, ... S2 C conf/st S3	**Array Status.** CP0 is running behavioral code of operator A (`merge`). CMB controller has set up appropriate active segment and operation mode for each CMB. On this diagram, CMB0 and CMB1 act as SeqSrc (sequential source) and CMB2 — SeqSink (sequential sink) relative to the connected streams. At this time, approximately half of tokens have been consumed from both sources CMB0 S0 and CMB1 S0 and sunk into CMB2 S1.
III	 CP0 Configuration: A Mode: Stall In 0 FIFO In 1 FIFO Logic/FSMs Out 0 FIFO Out 1 FIFO CMB0 Active Seg: S0 Mode: SeqSrc S0 0% 100% i0: S1 0% 100% i3: 4, 7, 7, ... S2 A conf/st S3 CMB1 Active Seg: S0 Mode: SeqSrc S0 0% 100% i1: S1 0% 100% S2 B conf/st S3 CMB2 Active Seg: S1 Mode: SeqSink S0 0% 100% S1 0% 100% t1: 2, 2, 3, ... S2 C conf/st S3 EMPTY EMPTY  2 CP0 Configuration: A Mode: Reconfig In 0 FIFO In 1 FIFO Logic/FSMs Out 0 FIFO Out 1 FIFO CMB0 Active Seg: S2 Mode: SeqSink S0 0% 100% i0: S1 0% 100% i3: 4, 7, 7, ... S2 A conf/st S3 CMB1 Active Seg: S2 Mode: SeqSrc S0 0% 100% i1: S1 0% 100% S2 B conf/st S3 CMB2 Active Seg: S1 Mode: SeqSink S0 0% 100% S1 0% 100% t1: 2, 2, 3, ... S2 C conf/st S3 1	*End of the first timeslice.* **Array Status.** All tokens from both sources CMB0 S0 and CMB1 S1 have been consumed by CP0 and sunk into CMB2 S1. If a source node of a stream is not producing any tokens (*e.g.* empty segment CMB0 S0), a sink node could stall due to unavailability of input tokens (*e.g.* CP0 is stalled, since operator A requires tokens on at least one input to fire). On the diagram such streams are identified with *EMPTY*. Scheduler uses the information about *EMPTY* streams to optimize schedule for the next timeslice. **Reconfiguration.** CP0 reconfiguration consists of two logically sequential steps, that could be parallelized if the array implementation permits. 1. Save current configuration and state of CP0 in CMB0 S2, which is allocated for A. 2. Load the configuration and state for page B into CP0 from CMB1 S2. After CP0 has been configured, the streams are created between compute nodes as shown on the next diagram. CP0 is ready to run.

continued on next page

continued from previous page

Time	Physical Array View	Command Description
IV		**Array Status.** CP0 is running behavioral code of operator B (`merge`). Approximately half of tokens in CMB0 S1 and CMB2 S1 have been consumed by CP0 and sunk into CMB1 S1.
V	 	*End of the second timeslice.* **Array Status.** All tokens from both sources CMB0 S1 and CMB2 S1 have been consumed by CP0 and sunk into CMB1 S1. If a sink node of a stream is not consuming tokens (*e.g.* 100% full CMB1 S1), a source node could stall on a stream write. On the diagram such streams are identified with *FULL*. Scheduler uses the information about *FULL* streams to optimize schedule for the next timeslice. **Reconfiguration.** Two main steps of reconfiguration are similar to those at time III. CP0 is loaded with configuration and state of page C (`uniq`).
VI		**Array Status.** CP0 is running behavioral code of operator C (`uniq`). Approximately half of tokens in CMB1 S1 have been consumed by CP0 and sunk into CMB2 S0.

continued on next page

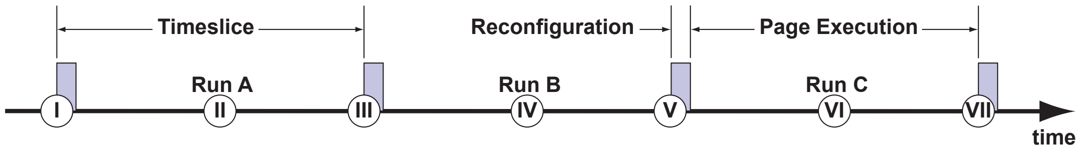
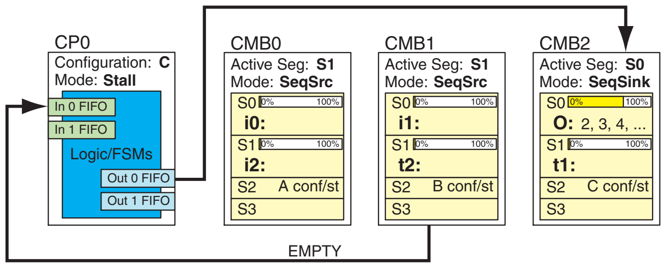
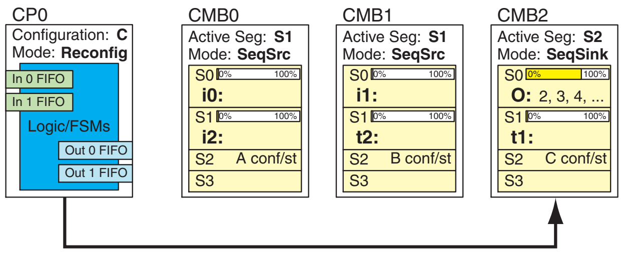


Figure 17: Timeline for Execution Example

continued from previous page

Time	Physical Array View	Command Description
VII	 CP0 · Configuration: C · Mode: Stall · In 0 FIFO · In 1 FIFO · Logic/FSMs · Out 0 FIFO · Out 1 FIFO · CMB0 · Active Seg: S1 · Mode: SeqSrc · S0 · i0: · S1 · i2: · S2 A conf/st · S3 · CMB1 · Active Seg: S1 · Mode: SeqSrc · S0 · i1: · S1 · t2: · S2 B conf/st · S3 · CMB2 · Active Seg: S0 · Mode: SeqSink · S0 · O: 2, 3, 4, ... · S1 · t1: · S2 C conf/st · S3 · 0% · 100% · EMPTY	*End of the third timeslice.* **Array Status.** All tokens from CMB1 S1 have been consumed by CP0 and sunk into CMB2 S0.
	 CP0 · Configuration: C · Mode: Reconfig · In 0 FIFO · In 1 FIFO · Logic/FSMs · Out 0 FIFO · Out 1 FIFO · CMB0 · Active Seg: S1 · Mode: SeqSrc · S0 · i0: · S1 · i2: · S2 A conf/st · S3 · CMB1 · Active Seg: S1 · Mode: SeqSrc · S0 · i1: · S1 · t2: · S2 B conf/st · S3 · CMB2 · Active Seg: S2 · Mode: SeqSink · S0 · O: 2, 3, 4, ... · S1 · t1: · S2 C conf/st · S3 · 0% · 100%	**Reconfiguration.** The current configuration and state of CP0 are saved in CMB2 S2. **Note:** For the application that was demonstrated here, saving configuration and state of CP0 was not necessary. A, B, and C were only scheduled once, and therefore after each one runs on the CP0 for a timeslice, its state is no longer needed. Saving of configuration and state was shown for completeness only. Should any of the pages be scheduled to run in several non-consecutive timeslices, their state must be saved every time they are preempted and restored when scheduled. This is analogous to context switching in traditional operating systems.

to produce master files (merge.cc and uniq.cc) and also parameterized instance files (merge_8.cc and uniq_8.cc). The next step is to compile all C++ sources including the driver code in main.C and the `tdfc`-generated sources. The build process terminates when all driver code is linked with the master files to produce user application executable (`a.out`), and instance objects are linked with the run-time system libraries to produce dynamically linked shared object libraries (merge_8.so and uniq_8.so) containing the instance code. The purpose of this process should become clear in the next section as we describe the SCORE run-time environment in more detail.

7.2 Run-time Environment

The run-time system consists of the scheduler and the simulator processes that execute under Linux as shown in Figure 19. In a real system, the OS kernel will contain the scheduler, and a reconfigurable hardware array will replace the simulator. These components are connected by a pair of streams that permit bidirectional communication and transmit scheduler commands and resource state to and from the array. The scheduler consists of instantiation and scheduling engines.

Instantiation engine. Being an independent process, the scheduler has no knowledge of user applications' compute graphs. The run-time system together with shared object files built with a user application provide a way to communicate the structure of compute graphs from a user application to the scheduler:

1. Upon invocation, a user application places a series of requests to the scheduler to instantiate its compute graph nodes. This is accomplished by the code in the master files, produced by `tdfc` and linked with the user executable. The code contains a sequence of operations to connect to the scheduler through an IPC channel and request to instantiate an operator. For example, in Figure 19 the code in the invoked `merge()` routine requests instantiation of the `merge` operator with inputs `t1` and `i2`.

2. With the request, the scheduler receives a pointer to the shared object file which contains the behavioral code and the attributes of a parameterized instance of an operator. The run-time system dynamically links with that shared object file (here, merge_8.so), and the scheduler instantiates an operator and places it on a waiting list to be scheduled. Note that the shared object is necessary here in order to get the user's application code loaded into the address space of the scheduler which, of course, was built without any knowledge of the user code which it might be asked to run.

   The array simulator executes the behavioral code for each resident compute node.

Scheduling engine. The scheduling engine is invoked every timeslice and is responsible for resource allocation and utilization, placement, and routing on the array. It acts as a resource manager capable of enforcing a variety of policies from fair sharing of the compute resources between multiple user applications to favoring a particular application to meet its real-time constraints.

Array simulator. The simulator provides a cycle accurate simulation by executing compute node behavioral code, found in corresponding dynamically linked shared object files (*e.g.* merge_8.so). As noted earlier, it communicates with the scheduler through a pair of streams. Implemented using shared memory, streams also provide direct communication between a user application and the array simulator. In the example in Figure 16 these streams are `i0`, `i1`, `i2`, and `o`.

8 Example: JPEG

As described in the previous section, we have implemented a complete SCORE run-time system and simulator on top of Linux and are beginning to develop several applications to guide our further understanding of critical design issues for these systems. As an early exercise and demonstration vehicle, we have implemented a complete JPEG (Joint Photographic Experts Group) image compression algorithm [33] in TDF and C++ and performed basic scaling experiments where we vary the number of computational pages in the system.

8.1 Application

The JPEG compressor mathematically decomposes the input data into high and low frequency components. The image is first segmented into 8×8 pixel blocks, and then the decomposition is performed on every individual block via the DCT (Discrete Cosine Transform), a unitary transform that takes the pixel block as an input and returns another 8×8 block of coefficients, most of which are close to zero. The coefficients are then scalar quantized and scanned into a one-dimensional stream via a *zigzag* scan. Quantized coefficients are subsequently compacted with zero-length encoding, after which runs and lengths are Huffman encoded. (See Figure 20.)

Our TDF implementation uses 13 512-LUT pages in order to realize a fully spatial JPEG compressor which is capable of processing one image sample per cycle.

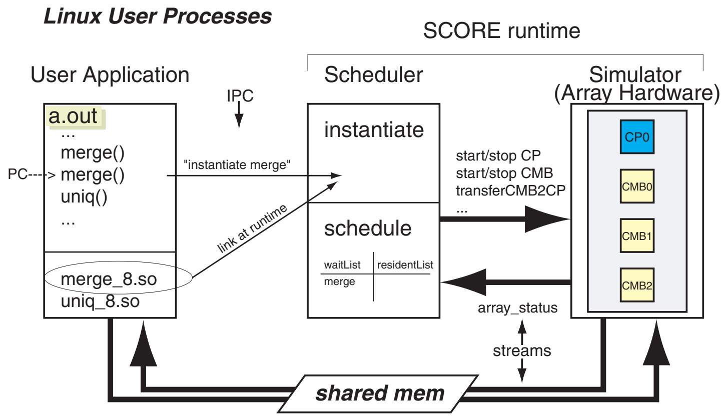
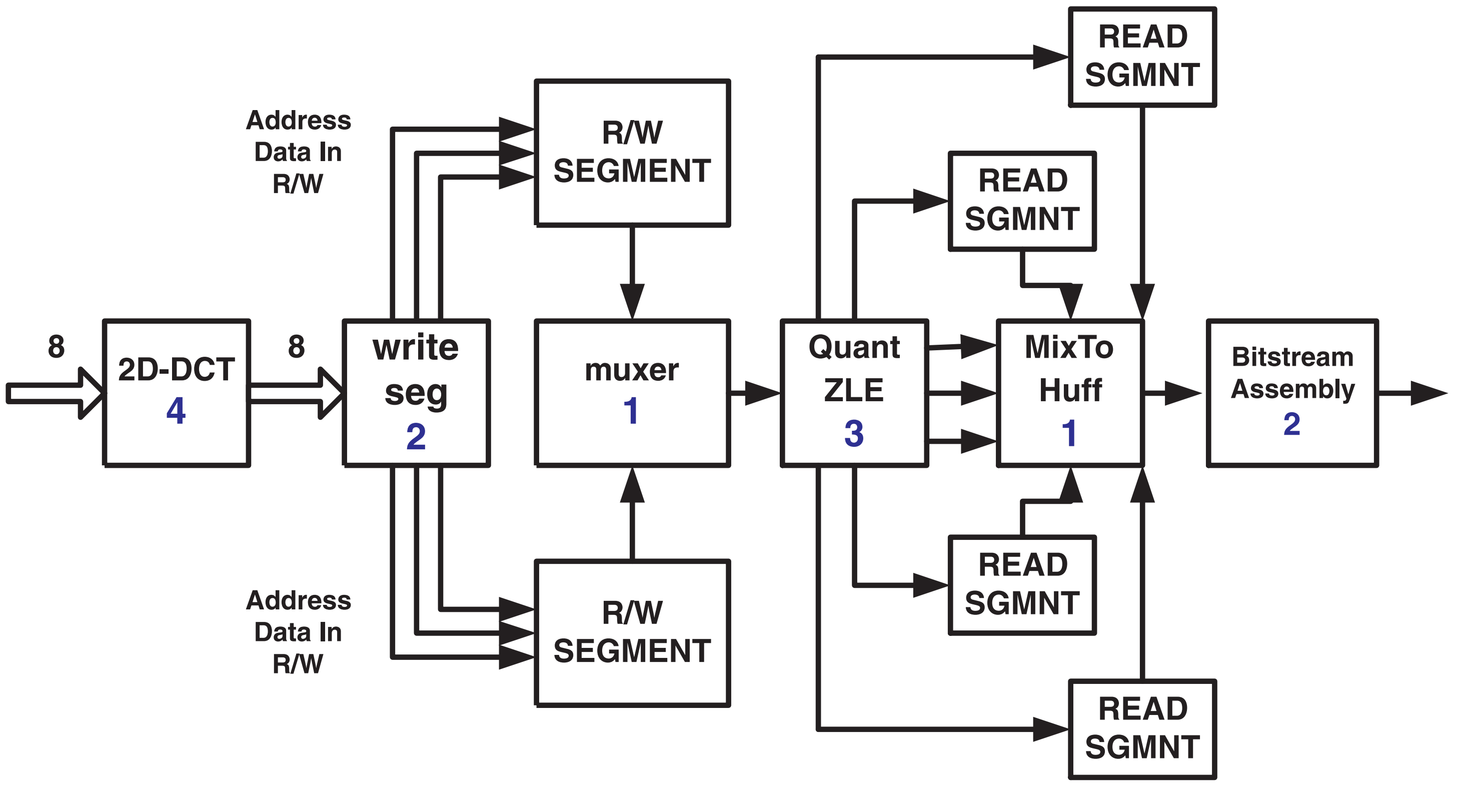
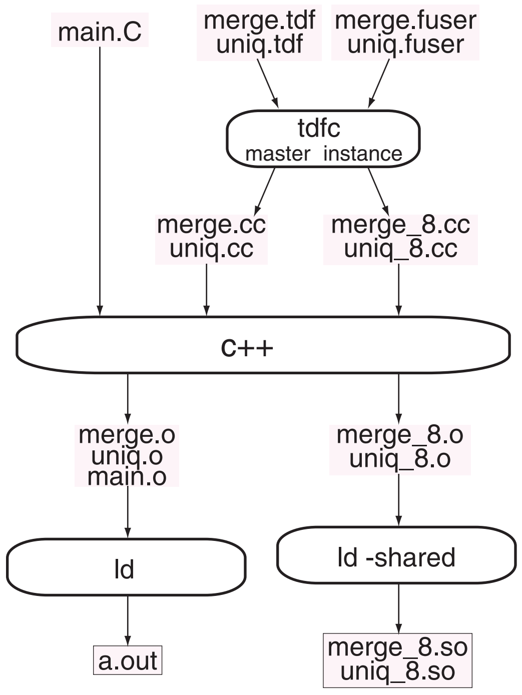


Figure 19: SCORE Run-Time Structure and Interaction to User Application



(*N.B.* Large numbers on computational blocks indicate number of 512-LUT pages required.)

Figure 20: JPEG Data Flow including Page and Segment Decomposition



Note: Each operator is described in a separate TDF source file. Tools used are `tdfc` (TDF compiler), `c++` (standard c++ compiler), and `ld` (standard linker with capability of building stand-alone executables and shared dynamically linked libraries).

Figure 18: Build Process for User Application from Fig. 16.

Simulator Parameters	**Value Assumed**
Reconfiguration Time	5,000 cycles
Schedule Time Slice	250,000 cycles
Compute Page (CP) size	512 LUTs
Configurable Memory Block CMB size	2Mbits
External Memory Bandwidth	2GB/s

Table 2: System Parameters for Experiment

.

For smaller hardware, the SCORE scheduler automatically manages, at run time, the reconfiguration necessary to share the physical CPs among the 13 virtual CPs.

8.2 System Assumptions

For these experiments, we assume a single-chip system as described in Section 4, with external memory as needed for the application. Table 2 summarizes the parameters we assume for the system, based on our experience with the HSRA [30] and embedded DRAM memory [25]. For these experiments page decomposition is performed manually. The scheduler is list based and operates in a time-sliced fashion like a conventional operating-system scheduler; the scheduler takes care of all decisions on where to place CPs and CMBs and manages all reconfiguration and data transfer, including the data movement on and off the component as necessitated by the finite, on-chip memory capacity. We assume scheduling time is overlapped with computation and takes 50,000 cycles. We do not, currently, model any limitations on routability among pages. The simulator accounts for all time required to reconfigure pages, store state, and transfer data between memories in the chip.

8.3 Results

To study the scalability, performance, and efficiency of SCORE, we ran our JPEG implementation on a series of simulated, architecture-compatible SCORE systems with varying numbers of physical compute pages. Figure 21 plots the total run time (makespan) of each system versus the number of physical pages in that system. In this particular experiment, we do not scale memory, so the results shown reflect (1) a fixed memory of 26 CMBs, and (2) unlimited memory. For comparison, we show a native x86-MMX implementation using Intel's reference *ijpeg* library.

The curves demonstrate that SCORE can automatically run the JPEG application on less hardware with graceful performance degradation. Thus SCORE can automatically realize an area-time performance tradeoff. Further, the curves show that this application can be automati-

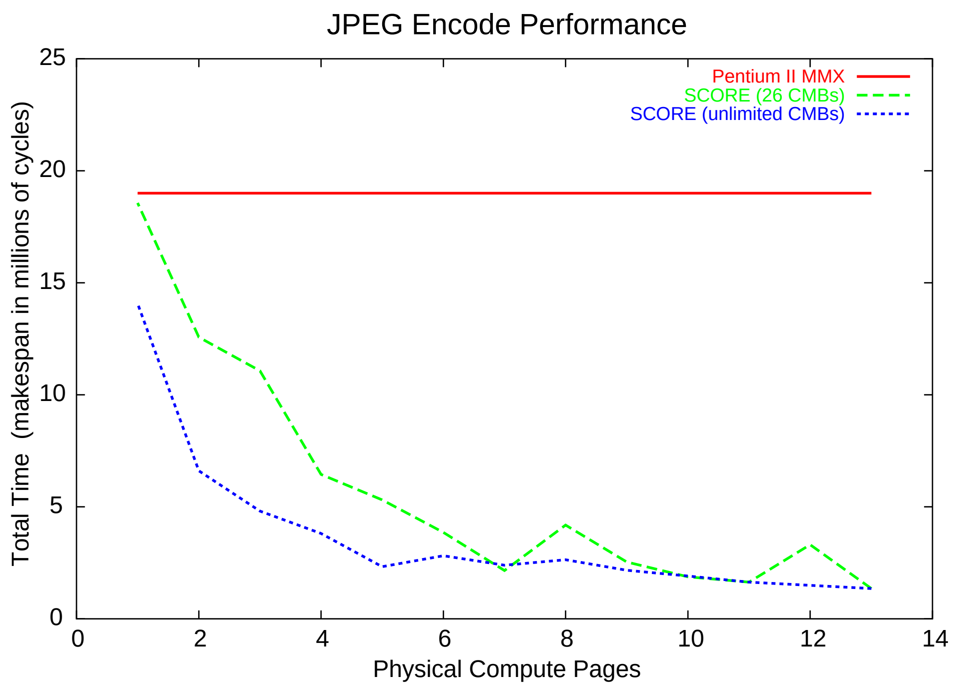


Figure 21: JPEG CP vs. Makespan

cally virtualized onto half the compute pages of the fully-spatial implementation without incurring a substantial performance penalty. This kind of result is common when the load on compute operators vary widely; the lightly-loaded operators can time-share a compute page without increasing overall runtime.

The experiment exhibits some anomalies of our present scheduler. The CP-makespan curves are not strictly monotonic due to heuristics in the list-based page selection approach. Also, the scheduler is not optimized to minimize memory usage while buffering streams. In fact, it is not possible to scale down the number of CMBs together with physical CPs in very small hardware because there would not be enough CMBs to virtualize the streams of presently-loaded pages. Hence, this experiment assumes a fixed memory availability of 26 CMBs (twice the number of CPs in the application). To factor out the effect of unoptimized stream buffering, we also performed the experiment with unlimited memory. The results exhibit a speedup of up to twofold over the limited memory case, suggesting that there is room for improvement in scheduling and memory management.

This experiment represents a single set of SCORE system parameters. As ongoing work, we are exploring many system parameters to gain insight into the regions of operation where SCORE scheduling is most robust and to determine the parameters that provide the most efficient and balanced system design. Such parameters include compute page size, page I/O bandwidth, memory block size, and reconfiguration times.

9 Summary

Reconfigurable computation, defined simply as computation performed on a collection of FPGA or FPGA-like hardware, has shown remarkable promise on point applications, but has not achieved wide-spread acceptance and usage. One must make a large commitment to a particular FPGA-based system to develop an application. However, as we can now readily predict, the industry produces newer, larger, and faster hardware at a steady pace. Unfortunately, without a unifying computational model which transcends the particular FPGA implementation on which the application is first developed, one is stuck redoing significant work to port the application to newer hardware. This is particularly onerous when the established, alternative technology, the microprocessor, offers users steady performance improvements with little or no time investment to adapt to new hardware.

Overcoming this liability requires a computational model which abstracts computational resources, allowing application performance to scale automatically, adapting to new hardware as it becomes available. The computational model must expose and exploit the strengths of reconfigurable hardware and help users understand how to optimize applications for reconfigurable execution. Further, the computational model must allow problems to deal efficiently with dynamic and unbounded resource requirements and dynamic program characteristics. Finally, the model must support the efficient composition of solutions from abstract building blocks.

In this paper, we have introduced a particular computational model which attempts to address these needs. SCORE uses a paging model to virtualize all hardware resources including computation, storage, and communication. It allows dynamic instantiation of dynamically sized computational operators and supports dynamic rate applications. A page partitioner and compiler along with a run-time scheduler takes care of automatically mapping the unbounded and dynamically unfolding computational graph onto the fixed resources of a particular hardware platform. We have outlined the hardware requirements for such a model as well as the kind of programming languages needed to describe and integrate SCORE computations. We have implemented a complete SCORE run-time system and simulator. Initial experiments suggest that we can achieve the desired scalability on sample applications. With this initial success, we are now attempting to broaden the range of applications, automate more of the SCORE tool flow, and systematically explore the design space for SCORE compatible architectures.

Acknowledgements

This research is part of the Berkeley Reconfigurable Architectures, Software, and Systems (BRASS) effort supported by the Defence Advanced Research Projects Agency under contract number DABT63-C-0048 and by the California MICRO Program.

References

[1] Arthur Abnous and Jan Rabaey. Ultra-Low-Power Domain-Specific Multimedia Processors. In *Proceedings of the IEEE VLSI Signal Processing Workshop (VSP'96)*, October 1996.

[2] Altera Corporation, 2610 Orchard Parkway, San Jose, CA 95134-2020. *APEX Device Family*, March 1999. `<http://www.altera.com/html/products/apex.html>`.

[3] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. *Software Synthesis from Dataflow Graphs*, chapter Synchronous Dataflow. Kluwer Academic Publishers, 1996.

[4] Vincent Michael Bove, Jr. and John A. Watlington. Cheops: A Reconfigurable Data-Flow System for Video Processing. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(2):140–149, April 1995. `<http://wad.www.media.mit.edu/people/wad/cheops_CSVT/cheops.html>`.

[5] Gordon Brebner. A Virtual Hardware Operating System for the Xilinx XC6200. In *Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications (FPL'96)*, pages 327–336, 1996.

[6] Gordon Brebner. The Swapable Logic Unit:a Paradigm for Virtual Hardware. In *Proceedings of the 5th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97)*, pages 77–86, April 1997.

[7] 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.

[8] Stephen P. Crago, Brian Schott, and Robert Parker. SLAAC: a Distributed Architecture for Adaptive Computing. In *Proceedings of the 1998 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'98)*, pages 286–287, April 1998.

[9] David E. Culler, Seth C. Goldstein, Klaus E. Schauser, and Thorsten von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, June 1993.

[10] Jack B. Dennis. Data Flow Supercomputers. *Computer*, 13:48–56, November 1980.

[11] Jack B. Dennis and David P. Misunas. A Preliminary Architecture for a basic data-flow processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture*, January 1975.

[12] Daniel Gajski and Loganath Ramachandran. Introduction to High-Level Synthesis. *IEEE Design and Test of Computers*, 11(4):44–54, 1994.

[13] Maya Gokhale, Janice Stone, Jeff Arnold, and Mirek Kalinoskwi. Stream-Oriented FPGA Computing in the Streams-C High Level Lanugage. In *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*. IEEE, April 2000.

[14] Seth C. Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. PipeRench: a Coprocessor for Streaming Multimedia Acceleration. In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA'99)*, pages 28–39, May 1999.

[15] Scott Hauck, Thomas Fry, Matthew Hosler, and Jeffery Kao. The Chimaera Reconfigurable Functional Unit. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96, April 1997.

[16] John R. Hauser and John Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the IEEE Symposium on Field-Programmable Gate Arrays for Custom Computing Machines*, pages 12–21. IEEE, April 1997.

[17] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[18] R. A. Iannucci. Toward a Dataflow/Von Neumann Hybrid Architecture. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 131–40, May 1988.

[19] Jeffery A. Jacob and Paul Chow. Memory Interfacing and Instruction Specification for Reconfigurable Processors. In *Proceedings of the 1999 International Symposium on Field Programmable Gate Arrays (FPGA'99)*, pages 145–154, February 1999.

[20] Mark Jones, Luke Scharf, Jonathan Scott, Chris Twaddle, Matthew Yaconis, Kuan Yao, and Peter Athanas. Implementing an API for Distributed Adaptive Computing Systems. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*, pages 222–230, April 1999.

[21] Edward A. Lee. *Advanced Topics in Dataflow Computing*, chapter Static Scheduling of Data-Flow Programs for DSP. Prentice Hall, 1991.

[22] X. P. Ling and H. Amano. WASMII: a Data Driven Computer on a Virtual Hardware. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'93)*, pages 33–42, April 1993.

[23] Bruce Newgard. Signal Processing with Xilinx FPGAs. `<http://www.xilinx.com/apps/appnotes/sd_xdsp.pdf>`, June 1996.

[24] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active Pages: a Model of Computation for Intelligent Memory. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA'98)*, June 1998.

[25] Stylianos Perissakis, Yangsung Joo, Jinhong Ahn, André DeHon, and John Wawrzynek. Embedded DRAM for a Reconfigurable Array. In *Proceedings of the 1999 Symposium on VLSI Circuits*, June 1999.

[26] Jan Rabaey. Reconfigurable Computing: The Solution to Low Power Programmable DSP. In *Proceedings of the 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'97)*, April 1997.

[27] Rahul Razdan and Michael D. Smith. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–180. IEEE Computer Society, November 1994.

[28] Paul T. Sasaki. A Fast FPGA (FFPGA) Using Active Interconnect. In *Proceedings of the 1998 International Symposium on Field-Programmable Gate Arrays (FPGA'98)*, page 255, February 1998.

[29] Edward Tau, Ian Eslick, Derrick Chen, Jeremy Brown, and André DeHon. A First Generation DPGA Implementation. In *Proceedings of the Third Canadian Workshop on Field-Programmable Devices*, pages 138–143, May 1995.

[30] William Tsu, Kip Macy, Atul Joshi, Randy Huang, Norman Walker, Tony Tung, Omid Rowhani, Varghese George, John Wawrzynek, and André DeHon. HSRA: High-Speed, Hierarchical Synchronous Reconfigurable Array. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 125–134, February 1999.

[31] John Villasenor, Chris Jones, and Brian Schoner. Video Communications using Rapidly Reconfigurable Hardware. *IEEE Transactions on Circuits and Systems for Video Technology*, 5:565–567, December 1995.

[32] John Villasenor, Brian Schoner, Kang-Ngee Chia, and Charles Zapata. Configurable Computer Solutions for Automatic Target Recognition. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 70–79. IEEE, April 1996.

[33] Gregory K. Wallace. The JPEG Still Picture Compression Standard. *Communications of the ACM*, 34(4):30–44, April 1991.

[34] John A. Watlington. MagicEight: An Architecture for Media Processing and an Implementation. Thesis proposal, MIT Media Laboratory, January 1999. <`http://wad.www.media.mit.edu/people/wad/tp/`>.

[35] John A. Watlington and V. Michael Bove, Jr. A System for Parallel Media Processing. In *Proceedings of the Workshop on Parallel Processing in Multimedia*, April 1997.

[36] Michael J. Wirthlin and Brad L. Hutchings. DISC: the Dynamic Instruction Set Computer. In *Proceedings of the SPIE Reconfigurable Computing Conference: Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing*, pages 92–103, October 1995.

[37] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *Virtex Series FPGAs*, 1999. <`http://www.xilinx.com/products/virtex.htm`>.

[38] Peixin Zhong, Margaret Martonosi, Pranav Ashar, and Sharad Malik. Accelerating Boolean Satisfiability with Configurable Hardware. In *Proceedings of the 1998 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'98)*, pages 186–195, April 1998.

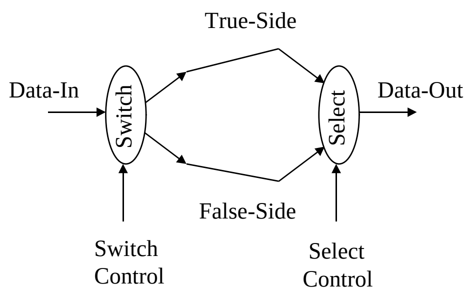


Figure 22: `switch`-`select` Example Motivating Unbounded Stream Buffers

A Unbounded Stream Buffers

All levels of the SCORE computational model provide the abstraction of unbounded stream buffers. Picking any finite buffer size for streams would introduce an artifact into the model which is very difficult to reason about. In particular, programs would be prone to deadlock any time the number of tokens which the application needed to queue up on a stream between a pair of operators was data dependent.

The canonical instance of this deadlock hazard is exemplified with a pair of nodes, `switch` and select. `switch` takes in two inputs, a boolean control stream, and a data stream. It sends its output along one of two output streams according to the value of the control input. `select` takes in three inputs, a control stream and two input streams. However, it does not read all three tokens on each cycle. Rather, it first reads the control token. Based on the value of the control token, it then reads from one of the two input streams and passes that along to its single output stream.

These two nodes can now be hooked up directly to each other with the two outputs of the `switch` node connected to the two inputs of the `select` node as shown in Figure 22. We provide separate control streams for the `switch` and `select` nodes. Now, if there is ever a stream prefix of the `switch`-control stream which contains n more TRUEs than FALSEs (or vice versa) than the `select`-control stream, the stream between the TRUE side of the `switch` and `select` nodes will have to hold n tokens. If the streams were limited to some fixed-size buffer m and $m < n$, then this subgraph would deadlock. Without loss of generality, consider the case in which the `switch` node received n TRUEs followed by one FALSE, while the `select` node initially receives one FALSE control signal followed by n TRUEs. The TRUE-side stream would fill up with m tokens. The `switch` node would not be able to perform any more operations because it cannot write data onto the TRUE-side stream. The `select` node, however, must process a token from the FALSE side in order to continue, but there are no tokens on the FALSE side to consume. The `select` node cannot make forward process until it is given a token on the FALSE side. The `switch` node cannot make any progress until the downstream operator (the `switch`) consumes a token on the TRUE side. These two operators are now deadlocked on a cyclic dependence. Note that if $m > n$ (or m unbounded), this deadlock would not occur and processing would be able to proceed.

Since, in general, these control streams can be completely independent, it is not possible to say that they will have any particular property between them. If these control streams were coming from outside of the system, we would certainly not have any control or knowledge of their relationship. Even if they were generated inside the system, the general question of whether or not a given computation produces a particular token value after a finite number of operations is equivalent to the halting problem.

Therefore, in order to provide reasonable semantics to the programmer, we accept the unbounded buffer size abstraction and include support in the execution model to expand finite buffers as necessary to meet this abstraction (up to the limit of the amount of memory we have available in the system).

B SCORE Compute Model

Section 3.1 described the compute model informally. This section defines it more precisely.

B.1 SCORE

SCORE computation is a graph, G:

$$\begin{array}{lcl} G & = & \{V, E\} \\ E & = & \{e_1, e_2, ...\} \\ e_i & \text{is a} & \text{SFIFO} \\ V & = & V_f \cup V_t \cup V_i \cup V_o \\ v_i & \in & V_f \text{ is a SFSM} \\ v_i & \in & V_t \text{ is a STM} \\ v_i & \in & V_i \text{ is a SIN} \\ v_i & \in & V_o \text{ is a SOUT} \end{array}$$

Notes:

- G will typically be initialized with at least one node $v_s \in V_t$ to start the computation.

- G may start with many nodes in V

B.2 SFIFO

SFIFO is a two-ended, unbounded FIFO which may be created, closed, and freed.

$$\begin{array}{rcl}
e & = & \{p_{src}, p_{sink}, Q, EOS, FREE\} \\
EOS & = & \text{Boolean} \\
FREE & = & \text{Boolean} \\
p_{src} & \text{is a} & \text{PORT} \\
p_{sink} & \text{is a} & \text{PORT} \\
Q & \text{is a} & \text{QUEUE}
\end{array}$$

Operations:
$v \in V \equiv$ vertex from which the operation is invoked.

Op	**Requirement**	**Action**
write(e,t)	$e.p_{src} \in \text{outs}(v)$ $t \in T_{data}$ $\overline{Q \rightarrow \text{eos}()}$	$Q \rightarrow$add(t)
close(e)	$e.p_{src} \in \text{outs}(v)$ $\overline{Q \rightarrow \text{eos}()}$	$Q \rightarrow$add(T_{EOS})
t=present(e)	$e.p_{sink} \in \text{ins}(v)$ FREE=false	t=$\overline{Q \rightarrow \text{empty}()}$
t=read(e)	$e.p_{sink} \in \text{ins}(v)$ FREE=false EOS=false	t=$Q \rightarrow$rm(); if ($t \equiv T_{EOS}$) EOS=true
t=eos(e)	$e.p_{sink} \in \text{ins}(v)$ FREE=false	t=EOS
free(e)	$e.p_{sink} \in \text{ins}(v)$ FREE=false	FREE=true

Notes:

- when an SFIFO is both freed and closed, it is removed from the SCORE graph. ($e.Q \rightarrow \text{eos}() \equiv e.$FREE $\equiv$ true $\Rightarrow E = E - \{e\}$)

B.3 QUEUE

QUEUE is a an unbounded queue.

$$\begin{array}{rcl}
Q.data & = & \text{ordered list of } T = \{q_0, q_1, q_2 ... q_{n-1}\} \\
 & = & \{\} \text{ when first created} \\
T & = & T_{data} \cup \{T_{EOS}\} \\
T_{data} & = & \text{finite alphabet} \\
Q \rightarrow \text{empty}() & \equiv & \text{value} = (\mid Q.data \mid \equiv 0) \\
Q \rightarrow \text{add}(t) & \equiv & Q.data = \{q_0, q_1, q_2 ... q_{n-1}, q_n = t\} \\
Q \rightarrow \text{rm}() & \equiv & \left\{ \begin{array}{ll} \overline{Q \rightarrow \text{empty}()} & \text{value} = q_0; \\ & Q.data = \{q_1, q_2, \\ & \quad ... q_{n-1}\} \\ Q \rightarrow \text{empty}() & \text{ERROR} \end{array} \right. \\
Q \rightarrow \text{eos}() & \equiv & \text{value} = (q_{n-1} \equiv T_{EOS})
\end{array}$$

B.4 SFSM

SFSM is an FSM with stream (SFIFO) I/O operations.

$$\begin{array}{rcl}
v_f & = & \{s_c, s_d, d, S_c, S_d, S_{res}, P_{in}, P_{out}, B\} \\
S_c & = & \{s_1, s_2, ... s_n\} \\
S_d & = & \text{also finite} \\
S_d & = & S_{res} \times S_{local} \\
s_c & \in & S_c \\
s_d & \in & S_d \\
d & \in & S_c \\
B & = & \{b_1, b_2, ... b_n\} \\
b_i & = & \{I_i, A_i, F_{ci}, F_{di}\} \\
I_i & \subset & P_{in} \\
A_i & = & \{a_{i,0}, a_{i,1}, ... a_{i,m_i}\} \\
a_{i,j} & \in & \{f_{i,j}, w_{i,j}, c_{i,j}\} \\
cs_d & = & \text{current data state} \in S_d \\
w_{i,j} & = & \text{if } (g_{i,j}(cs_d)) \text{ write}(p_{out} \in P_{out}, v_{i,j}(cs_d)) \\
c_{i,j} & = & \text{if } (g_{i,j}(cs_d)) \text{ close}(p_{out} \in P_{out}) \\
f_{i,j} & = & \text{if } (g_{i,j}(cs_d)) \text{ free}(p \in P_{in}) \\
v_{i,j}(cs_d) & = & F : S_d \rightarrow T_{data} \\
g_{i,j}(cs_d) & = & F : S_d \rightarrow \text{Boolean} \\
F_{ci} & = & F : S_d \rightarrow S_c \\
F_{di} & = & F : S_d \rightarrow S_d \\
I_d & = & \{\} \\
A_d & = & \{\} \\
F_{cd} & = & F : S_d \rightarrow \{d\} \\
F_{dd} & = & cs_d \text{ (Identity)} \\
d & = & \{I_d, A_d, F_{cd}, F_{dd}\}
\end{array}$$

Operation:

1. **Read:** in state s_i, if all inputs in I_i are present, read inputs into present state;[7] otherwise, do nothing (stay in state, perform no actions or transitions).
2. **Action:** perform all guarded writes, closes, and frees in A_i whose guards are true.
3. **Transition:** update state according to F_{ci} and F_{di}.

Notes:

- SFIFO operations present and read are only available to the read mechanism; they are not available for arbitrary use within the SFSM.
- d is the done state; an SFSM is done when it enters d.

[7] *N.B.* values of present state, cs_d, actually change to reflect input values.

- A well formed SFSM will close all output streams and free all input streams before making final transition to d.
- A properly specified SFSM will specify both the EOF and data transitions; on EOF of an input, it should transition only to a state that does not read that input and which cannot reach any state which can read said input.
- In a properly formed SFSM after performing a close action on an output, the machine will transition to a state which cannot reach any state which performs a write or close to said output.
- Separating S_c and S_d is artificial, but introduced for clarity.
- One might want to think of every state $s_c \in S_c$ being the multi-state sequence **Read**, **Action**, **Transition**.
- Strictly speaking, a memory segment (in a particular mode of operation) is an SFSM; From a formal standpoint, it is not necessary to define them as separate entities.
- During execution of an SFSM cs_d can only be seen or modified by the SFSM. After entering the done state, $s_{res} \in S_{res}$ is available to other operators (see below). Notably, for segments, the contents of the memory segment, M_i, is the resulting s_{res}.
- close(p) corresponds to e$\rightarrow$close() where e is the edge of which p is the source port; free and write have a similar correspondence.
- When an SFSM enters the done state it may be removed from the SCORE compute graph and any memory segments it owns are reverted to the creating vertex ($V = V - \{v_f\}$, $M_i.O = v_c$ where v_c allocated v_f).

B.5 PORT

A port is simply a designation of an input or output from a computational node. It is where the stream edge and the vertex actually come together.

B.6 SIN

SIN is an input stream from outside of the SCORE computational graph.

$$v_i = \{D_{src}, p_{sink}\}$$

Data arriving from D_{src} are placed as tokens into the SFSM attached at p_{sink}. The policy for the conversion is outside of the computational model, this simply represents the edge of the compute model (any algorithmic handling needed at the boundary can be represented with an SFSM).

B.7 SOUT

SOUT is an output stream from outside of the SCORE computational graph.

$$v_o = \{p_{src}, D_{sink}\}$$

Data arriving from the stream attached at p_{sink} is exposed to the outside at D_{sink}. Provisions will have to be made to signal consumption off of the associated SFIFO. Again, the policy for the conversion is outside of the computational model, this simply represents the edge of the compute model.

B.8 SMEM

An SMEM is a single-owner, finite-sized, random-access memory segment.

$$\begin{aligned} m &= \{D, O, A\} \\ sz &= \text{finite integer} \\ D &= (T_{data})^{sz} \\ O &\in V \cup \{\emptyset\} \\ A &\in V \end{aligned}$$

Operations:
$v \in V \equiv$ vertex from which the operation is invoked.

Op	**Requirement**	**Action**
alloc(t)	–	return new M with $sz = t$ with $O = v$
free(m)	$v \equiv A$	while($m.O \neq v$) {}; $m.O = \emptyset$
write(a,t,m)	$m \in v.M$ $t \in T_{data}$ $0 \leq$ a $\leq sz - 1$	while($m.O \neq v$) {}; $m.D$[a] $=$ t

t=read(a,m)	$m \in v.M$
	$0 \leq$ a $\leq sz - 1$
		while($m.O \neq v$) {};
		t=$m.D$[a]

Notes:

- This describes a model which only has one kind of lock (exclusive ownership). It would be relatively straight forward to define a model which allowed multiple readers.

B.9 STM

An STM is a Turing complete vertex with:
1. stream operations
2. ability to create SCORE graph nodes and edges and add them to the SCORE computational graph G

3. ability to lock a region of memory and block waiting on access to a region of memory

Mostly this can be defined as a superset of the SFSM where we add:

- memory allocation to allow unbounded memory in the vertex
- additional actions for new items 2 and 3 above.

$$
\begin{aligned}
v_t &= \{s_c, s_d, d, S_c, S_d, M, P_{in}, P_{out}, B\} \\
S_c &= \{s_1, s_2, ...s_n\} \\
S_d &= \text{also finite} \\
M &= \{M_1, M_2, ...M_m\} \\
s_c &\in S_c \\
s_d &\in S_d \\
d &\in S_d \\
B &= \{b_1, b_2, ...b_n\} \\
b_i &= \{I_{mi}, I_{si}, A_i, F_{ci}, F_{di}\} \\
I_{mi} &\subset M \\
I_{si} &\subset P_{in} \\
A_i &= \{a_{i,0}, a_{i,1}, ...a_{i,k_i}\} \\
a_{i,j} &\in \{f_{i,j}, w_{i,j}, c_{i,j}, am_{i,j}, fm_{i,j}, wm_{i,j}, \\
&\qquad ne_{i,j}, ng_{i,j}\} \\
cs_d &= \text{current data state} \in S_d \\
w_{i,j} &= \text{if } (g_{i,j}(cs_d)) \text{ write}(p_{out} \in P_{out}, v_{i,j}(cs_d)) \\
c_{i,j} &= \text{if } (g_{i,j}(cs_d)) \text{ close}(p_{out} \in P_{out}) \\
f_{i,j} &= \text{if } (g_{i,j}(cs_d)) \text{ free}(p \in P_{in}) \\
am_{i,j} &= \text{if } (g_{i,j}(cs_d)) \text{ alloc}(v_{i,j}(cs_d)) \\
fm_{i,j} &= \text{if } (g_{i,j}(cs_d)) \text{ free}(M_k \in M) \\
wm_{i,j} &= \text{if } (g_{i,j}(cs_d)) \\
&\qquad \text{write}(v_{i,j_1}(cs_d), v_{i,j_2}(cs_d), M_k \in M) \\
ne_{i,j} &= \text{if } (g_{i,j}(cs_d)) \text{ alloc\_sfifo}() \\
ng_{i,j} &= \text{if } (g_{i,j}(cs_d)) \\
&\qquad \text{alloc\_vertex}(VP, v_{i,j_1}(cs_d), \\
&\qquad\quad v_{i,j_2}(cs_d), ...v_{i,j_n}(cs_d), E_{si,j}, M_{si,j}) \\
VP &\text{ is a } \text{vertex prototype} \\
&\qquad (\text{SFSM or STM definition}) \\
E_{si,j} &= \{e_{si,j_1}, e_{si,j_2}, ..., e_{si,j_{k_{i,j}}}\} \mid e_{si,j_l} \in E \\
M_{si,j} &= \{m_{si,j_1}, m_{si,j_2}, ..., m_{si,j_{o_{i,j}}}\} \mid m_{si,j_l} \in M \\
v_{i,j_l}(cs_d) &= F : S_d \to T_{data} \\
v_{i,j}(cs_d) &= F : S_d \to T_{data} \\
g_{i,j}(cs_d) &= F : S_d \to \text{Boolean} \\
F_{ci} &= F : S_d \to S_c \\
F_{di} &= F : S_d \to S_d
\end{aligned}
$$

$$
\begin{aligned}
I_d &= \{\} \\
A_d &= \{\} \\
F_{cd} &= F : S_d \to \{d\} \\
F_{dd} &= cs_d \text{ (Identity)} \\
d &= \{I_d, A_d, F_{cd}, F_{dd}\}
\end{aligned}
$$

Operation:

1. **Read:** in state s_i, if all inputs in I_{si} are present and all memories I_{mi} are owned by this vertex, read inputs into present state;[8] otherwise, do nothing (stay in state, perform no actions or transitions).
2. **Action:** if all M_i's written by writes or freed be memory frees in A_i are owned by this vertex, perform all guarded writes, frees, closes, allocates, news in A_i whose guards are true; otherwise, do nothing (stay in this substate, perform no actions or transitions).
3. **Transition:** update state according to F_{ci} and F_{di}.

Actions:

SMEM and SFIFO operations are as previously defined. close(p) corresponds to e→close() where e is the edge of which p is the source port; free and write have a similar correspondence.

Op	Requirement	Action
alloc_sfifo()	–	returns an empty SFIFO, ne, with source and sink unbound; $E = E \cup \{ne\}$
alloc_vertex (t$_1$,t$_2$,..., E_s,M_s)	$E_s \subset E$ $M_s \subset M$ VP defined args match VP $t_i \in T_{data}$	while $(\exists m_i \in M_s \mid m_i.O \neq v)\{\}$; nv= new VP $\forall_{m_i \in M_s} (m_i.O = nv)$ set E_s sources and sinks to ports in nv set initial state in nv based on t_i's $V = V \cup \{nv\}$

Notes:

- SFIFO operations present and read are only available to the read mechanism; they are not available for arbitrary use within the STM. Similarly the SMEM operation read is only available to the read mechanism.
- d is the done state; an STM is done when it enters d.
- A well formed SFSM will close all output streams and free all inputs before making final transition to d.
- A properly specified SFSM will specify both the EOF and data transitions; on EOF of an input, it should transition only to a state that does not read that input and which cannot reach any state which can read said input.
- In a properly formed SFSM after performing a close action on an output, the machine will transition to a state

[8] *N.B.* values of present state, cs_d, actually change to reflect input values.

which cannot reach any state which performs a write or close to said output.

- Separating S_c and S_d is artificial, but introduced for clarity.
- One might want to think of every state $s_c \in S_c$ being the multi-state sequence **Read**, **Action**, **Transition**.
- When an STM enters the done state it is removed from the SCORE compute graph and any memory segments it owns are reverted to the creating vertex ($V = V - \{v_t\}$, $M_i.O = v_c$ where v_c allocated v_f).