Dynamic Runtime Scheduler Support for SCORE

By Michael Monkang Chu

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor John Wawrzynek
Research Advisor

(Date)

****

Professor John Kubiatowicz
Second Reader

(Date)

Table of Contents

Table of Figures

Table of Tables

Abstract

Reconfigurable computing devices offer substantial improvements in functional density and yield versus traditional microprocessors, yet remain out of general-purpose use due in part to their difficulty of programming and lack of cross-device compatibility. In [CASPI00] a stream-based compute model called SCORE (Stream Computations Organized for Reconfigurable Execution) was introduced with a goal to provide a programming model for easily exploiting the computational density of reconfigurable devices. SCORE virtualizes reconfigurable 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 efficiently run on a wide range of hardware. In this project we implemented project implements a dynamic runtime scheduler for SCORE that virtualizes the reconfigurable computation fabric and automatically manages the execution of SCORE applications in hardware. Initial performance scaling experiments show that a dynamic scheduler is able to automatically scale applications on reduced hardware and exploit hardware under-utilization to achieve reasonable area-time curves. In this paper, we present the basic scheduler details and runtime system flow along with key implementation highlights, such as scheduling heuristics, memory management, and deadlock detection.

1 Introduction

A reconfigurable device is a programmable semiconductor chip containing an array of configurable logic blocks and interconnects. These logic blocks and interconnect can be configured to perform computations by physically loading instruction bit-streams into the array. By loading a different instruction bit-stream, these devices can be reconfigured to perform a different computation.

Reconfigurable devices have proven extremely efficient for certain types of processing tasks. The key to their cost/performance advantage is that conventional processors are often limited by instruction bandwidth and execution restrictions or by an insufficient number or type of functional units. Reconfigurable logic can exploit more program parallelism. By dedicating significantly less instruction memory per active computing element, reconfigurable devices can achieve a 10x improvement in functional density over microprocessors. At the same time, this lower memory ratio allows reconfigurable devices to deploy active capacity at a finer grained level, allowing them to realize a higher yield of their raw capacity, sometimes as much as 10x versus conventional processors ([DEHON96]).

While reconfigurable devices can be used in isolation, there is also increasing interest in hybrid architectures such as DPGA ([DEHON96]) and Garp ([HAUSER97]), coupling reconfigurable logic (FPGAs) with a general-purpose processor (RISC). This allows applications to specialize the reconfigurable hardware to match application requirements while allowing operations that run inefficiently on the reconfigurable device to execute on the processor. In addition, the processor is available as a platform for managing the reconfigurable device and providing common operating system support such as file system access.

Another important development in reconfigurable devices is high-density embedded DRAM such as in the HSRA ([PERISAKISS99], [TSU99]). Devices available in the market today provide limited amounts of fine-grain SRAM, currently up to 96 blocks of 4 Kbits[1]. This is insufficient for the data sets of many applications, making it necessary to manage the internal memory as a cache and store the full data set in external memory, accessible through a low bandwidth external interface. In addition, when a reconfigurable device needs to be configured "on the fly", configuration time can be limited by the fact that configuration bitstreams reside in external memory. If, on the other hand, bitstreams are preloaded into internal memory, bandwidth and latency to the configuration is increased by orders of magnitude, making rapid dynamic reconfiguration possible. DRAM, with an order of magnitude higher density than SRAM, solves both of these problems by providing the ability to integrate large memory blocks on-chip.

Despite the advantages in cost/performance and developments in architecture and memory densities, reconfigurable devices still exist mainly as application-specific devices, out of reach of typical software programmers. One of the reasons is the lack of convenient programming tools and environments. Another reason preventing reconfigurable devices from gaining acceptance is that attaining their performance/cost advantages often requires exposing the underlying hardware to the programmer, making user programs device-dependent.

SCORE [CASPI00] attempts to solve both of these issues by providing a coherent model for expressing both processor and array computations in a way that can be easily mapped onto a reconfigurable array. With the help of a runtime system providing a infinite hardware abstraction, applications can be automatically run on SCORE-compliant hardware of various sizes. The runtime system contains a dynamic scheduler responsible for accepting a user designs and executing them on the hardware to completion.

In the following section, a brief introduction to the SCORE compute model is presented. The remainder of this paper is dedicated to explaining the details of the current implementation of a dynamic SCORE scheduler and runtime system. Initial performance scaling results are given to show that automatic scheduling for reconfigurable arrays is able to achieve acceptable area-time curves when presented with reduced hardware. Finally, the paper concludes with a discussion of future work and a conclusion.

1.1 SCORE Compute Model[2]

A compute model defines the computational semantics that a developer expects the hardware to provide. For convenience, the SCORE compute model is best viewed at two levels of abstraction. The *execution model* defines the run-time view of a SCORE computation. That is, it defines the run-time data structures used to define a SCORE computation as well as how the hardware will dynamically interpret this

[1] Xilinx XCV1000

[2] Adapted from [CASPI00]

description. The *programming model* provides a higher level view of SCORE application composition and execution suitable for the programmer. It abstracts away some of the hardware size details visible in the execution model, focusing the programmer on the style of computation and program composition suitable for SCORE execution. In this report, only the execution model is discussed; consult [CASPI00] for details on the programming model.

1.1.1 Execution Model

For any programmable computing architecture, we will need a language or format for describing the computation that the computer is to perform. The key idea of computer architecture is that it defines the computational description that a machine will run (i.e. 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 same description of the computation (i.e. i80286, i80386, i80486, Pentium, and K6 are all different implementations which conform to the x86 ISA architectural definition and all run the same computational descriptions). Following this technique, the execution model for SCORE defines the run-time description of a computation for an architecture family and the semantics expected for executing this description.

The SCORE execution model includes the following key components:

?? Fixed-size *compute page* (CP) – a block of reconfigurable logic that is the basic unit of virtualization and scheduling.

?? *Memory segment* – a contiguous block of memory which is the basic unit for data page management.

?? *Stream link* – a mechanism for logically connecting the output of one node (CP, segment, processor, or IO) to another node.

The run-time description defines all computations in terms of these basic building blocks. This description 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 semantics provided by the architecture is that of an unlimited number of independently operating physical compute pages and memory segments. Compute pages and memories operate on stream data tagged with input presence and produce output data to streams in a similar manner. The use of presence tags provides an operational semantics that is independent of the timing of any particular SCORE-compatible computing platform.

1.1.1.1 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 that is mapped onto physical hardware and managed as an atomic entity. Each compute page represents a fixed-size piece of reconfigurable hardware (i.e. 64 4-LUTs).

The compute page decomposition takes the stand that it is neither feasible nor desirable to manage every primitive computational building block (i.e. 4-LUT) as an independent entity—just as it is generally not desirable to manage every bit of memory as an independent memory. 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 within a page, to be performed offline within the page. Note that it is necessary that the page size be fixed across an architecture family so that all family members 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 (see Figure 1).

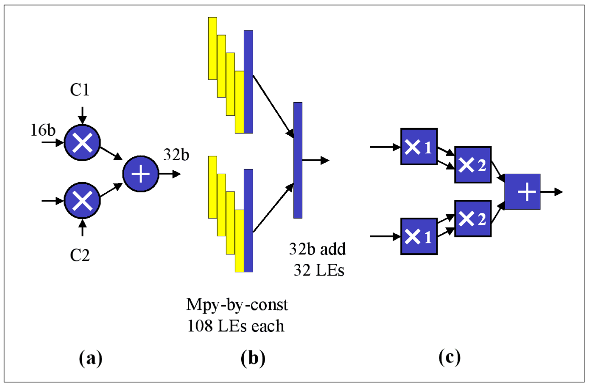


Figure 1 - Example of Page Decomposition: (a) original operator, (b) mapped to logic elements (LEs), (c) decomposed into fixed-size, 64-LE pages

Compute pages may contain internal state. Since the semantics provided by the hardware is that of an unbounded number of compute pages, the state associated with a CP must be saved and restored when a CP is swapped off of and on to a physical compute page.

1.1.1.2 Memory Segments and Configurable Memory Blocks

A *memory segment* is a contiguous block of memory that is managed as a single, atomic memory block for the purposes of swapping and relocation. Memory segments can be used in several modes (i.e. FIFO, read-only, random read-write). When configured into a particular operating mode, a segment will have its own stream ports (i.e. address input, data input, data output, control input) which connect it into the computational graph of pages and segments (see Figure 2).

To use a memory segment, the run-time system will map it into a *configurable memory block* (CMB) (see Figure 3). The CMB is a physical memory block inside the reconfigurable array with active stream links and interconnect to connect the memory segment into the live 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 (see Figure 3). 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. In our current vision, only a single such segment may actually be live at any point in time, but there is nothing in the SCORE definition that prevents an implementation from being designed to handle multiple, live segments in the same CMB.

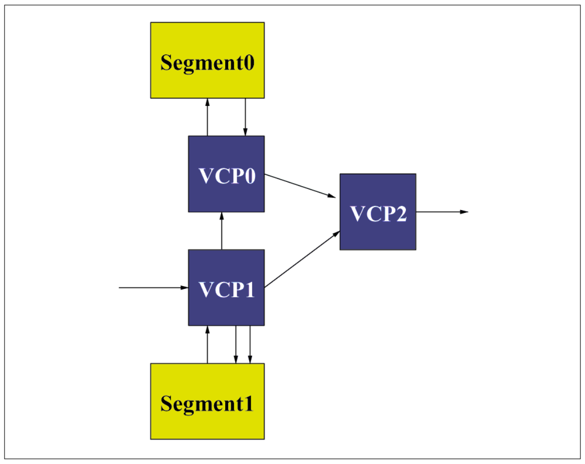


Figure 2 - Dataflow Computation Graph with both Compute Pages and Segments

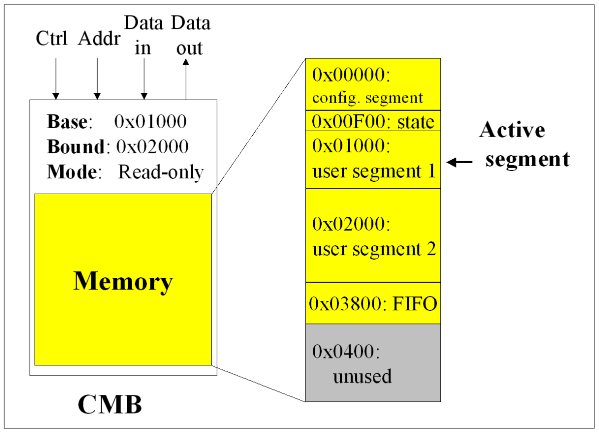


Figure 3 - Segments and Other Data mapped onto a CMB

1.1.1.3 Physically Finite, Logically Unbounded Streams

Streams form the data flow links between pages. A node (CP or segment) indicates when it is producing a valid data output with an out-of-band data present bit. The data value (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 4).

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.

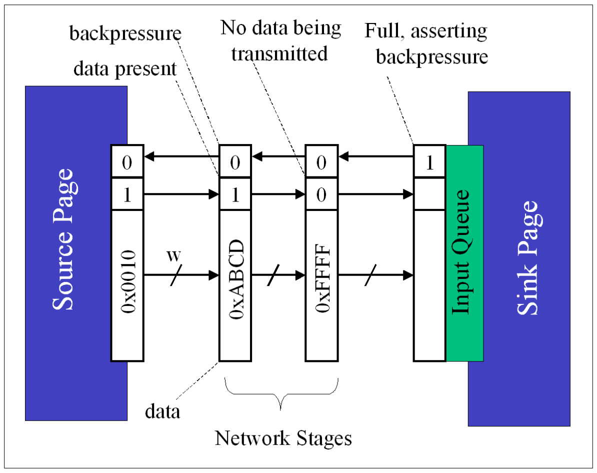


Figure 4 - Stream Signals

Even at the run-time level, these streams provide the abstraction of unbounded capacity links between producers and consumers. 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 4) to stall production of data items and the run-time system will allocate additional buffer space in the form of FIFO segments as needed.

Physically, a stream may be realized in two ways:

?? When both the producer and the consumer of a vstream are instantiated on the physical hardware, the stream link can be implemented as a spatial connection through the inter-page routing network between the two pages. (See Figure 6)

?? 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 7)

This allows efficient, pipelined chaining of co-resident operators when space permits, as well as deep intermediate data buffering when it is necessary to sequentialize computation.

1.1.1.4 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 must handle the scheduling of 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 6 shows this case for the video processing operator of Figure 5.

If hardware resources are limited, a computation graph will be time-multiplexed onto the hardware. Streams between virtual pages which are not simultaneously loaded will be transparently buffered through CMBs. Figure 7 shows this case for the video processing operator. Each component operator is loaded into hardware in sequence, taking its input from one CMB and producing its output to another. Configuration information and user data for swapped out pages are also stored in CMBs when the page is not resident.

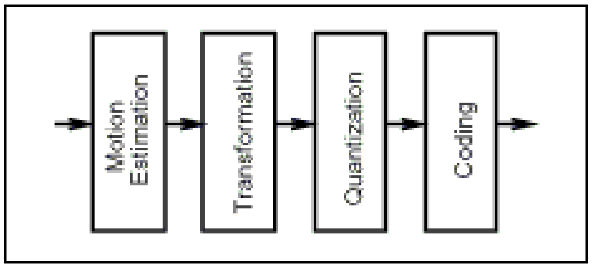


Figure 5 - Video Processing Operator

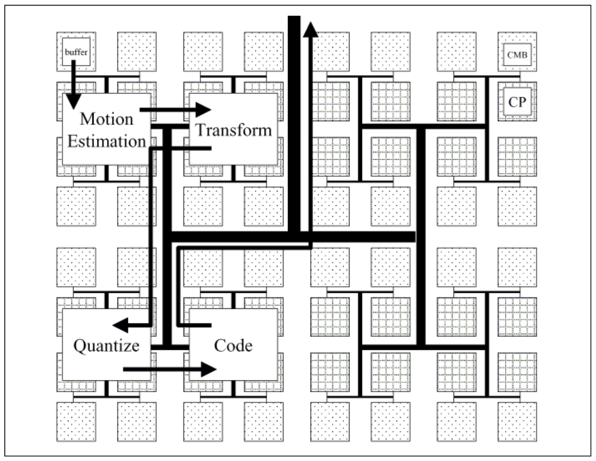


Figure 6 - Fully Spatial Implementation of Video Processing Operator

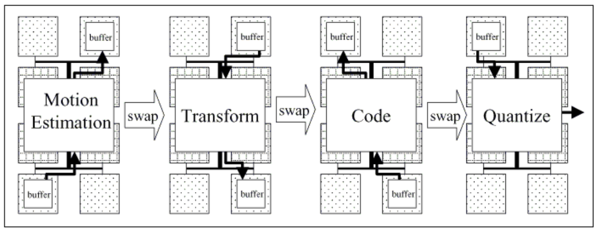


Figure 7 - Capacity-Limited, Temporal Implementation of Video Processing Operator

1.1.2 Hardware Requirements

SCORE assumes a combination of a sequential processor and a reconfigurable device. Although more stylized than simply placing an FPGA on an expansion bus, the requirements for a SCORE implementation are fairly modest. The reconfigurable array must be divided into a number of equivalent and independent compute pages. Multiple, distributed memory blocks are required to store intermediate data, page state, and page configurations.

The interconnect between pages must:

?? Provide adequate bandwidth to memory, allowing different memory pages to be used concurrently,

?? Support high bandwidth, low latency communication between active compute pages,

?? Provide buffering for pipelining data, and a back-pressure signal to stall upstream computation when the network buffer capacity is exceeded,

?? Provide sufficiently rich interconnect to facilitate rapid, online routing.

The compute pages themselves can be designed using any reconfigurable fabric as long as there is support for rapid reconfiguration. This support should include the ability to save as well as restore array state quickly. Although configuration caches may be beneficial (i.e. [TAU95], [HAUSER97]), we anticipate a wide range of applications where microsecond reconfiguration times are adequate for good performance. The subarray design from the HSRA [TSU99] is a feasible concrete implementation for a compute page. It provides microsecond reconfiguration and high-speed, pipelined computation. The symmetry of these compute blocks allows a single virtual compute page configuration to run on any physical compute page in the array.

Each configurable memory block (CMB) is a self-contained unit with its own address interface, data path, and address generator. Hence CMBs may be accessed independently and concurrently in a scalable system. The CMB can be an external RAM component or an on-chip memory bank (i.e. BRASS Embedded DRAM [PERISSAKIS99]) with logic to tie it into the data flow synchronization used by the interconnect network. The memory controllers need to support a simple paged segment model, allow the scheduler to relocate memory blocks within a physical memory page, and provide protection via segment bound registers. Since streaming access is commonly used during reconfiguration, state swapping, and stream buffer operations, dedicated stream access modes are useful to minimize external address bandwidth requirements.

A SCORE-ready reconfigurable array must also support several out-of-band signals. These signals are used by the dynamic runtime scheduler to query the status of the executing CPs and CMBs. The scheduler uses the runtime status to evaluate the effectiveness of a particular design mapping. A simple scheduler requires the following basic signals:

?? Each CP and CMB must have a concept of done to indicate it is done processing data.

?? Each CP and cMB needs to keep track of how often it is stalled on input underflows and output overflows.

?? Each CP and CMB must be able to report which I/O streams cause the current stall.

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 which would not run efficiently in reconfigurable implementation. Both of these functions require that the processor be able to control and communicate with the array efficiently. A single-chip SCORE system (i.e. see Figure 8) integrating a processor, reconfigurable fabric, and memory blocks could provide tight, efficient coupling of components.

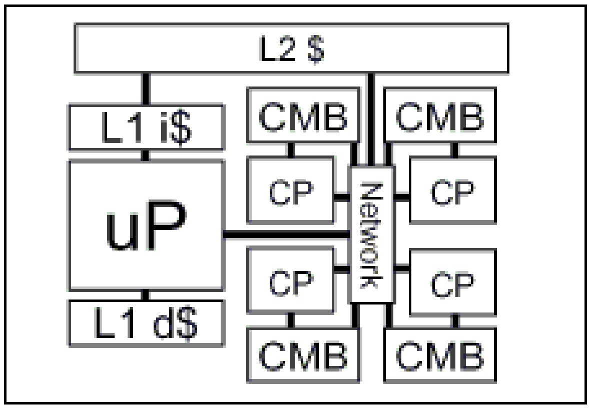


Figure 8 - Hypothetical, single-chip SCORE system

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.

1.2 Scheduler Responsibilities

The SCORE scheduler is one of the key components in the runtime system. It is responsible for three main tasks:

?? It accepts operators partitioned into fixed-size pages and segments from the user program and schedule the design to completion;

?? It manages all of the hardware resources on the array, including the CPs, CMBs and routing resources;

?? It provides the functional abstraction of infinite hardware to the application.

In addition to these responsibilities, the scheduler must also guarantee that deadlock and bufferlock are not introduced into the design. Also, livelock, or starvation, should be avoided by the scheduling heuristic. In Section 2, we show how the current SCORE implementation fulfills these responsibilities.

1.3 Related Work

1.3.1 Multiprocessors

SCORE shares with the multiprocessor community the notions of *priority-list scheduling* [GAJSKI92] and *gang scheduling* [FRANKE96]. Given a task precedence graph, priority-list scheduling uses a priority function to choose from among all tasks whose predecessors have completed, which to schedule. In the SCORE model, priorities can be used in conjunction with a dataflow graph to choose among operators whose predecessors have fired. Priorities may include, for instance the number of input tokens queued, and whether an operator configuration is already loaded on the array. [LIAO94] found that no single priority heuristic was optimal across different program structures and multiprocessor configurations, but that adaptive combinations thereof produced good results. Gang scheduling involves co-scheduling related tasks. In the SCORE model, it is clearly advantageous to co-schedule neighboring operators from the dataflow graph. In addition, it is highly advantageous to co-schedule all operators belonging to a feedback loop to avoid context swaps in each traversal of the loop.

There are some hardware similarities between a SCORE-based reconfigurable array and traditional message-passing multiprocessors. The array consists of nodes containing a processor with memory, communicating via point-to-point paths on a fat-tree network. Each node, however, is much smaller than a microprocessor. A CP containing 64 dual-4-LUT blocks, for instance, is comparable in complexity to an ALU. Such small nodes necessitate centralized control (on an external processor) for context swapping and job scheduling. In addition, the array has operating costs different from multiprocessors. The streaming capabilities of the network make inter-page communication relatively cheap since pipelining can hide network latency. Context swaps, which cost hundreds to thousands of cycles, thus lead to very high amounts of lost computation. The disparity in cost between communication and context swaps is thus far more extreme than in multiprocessors, where tasks are longer lived, and communication (which may require kernel intervention) has cost more comparable to context swaps.

1.3.2 Dataflow Systems

Because fully dynamic, run-time scheduling can be prohibitively expensive, various efforts appear in multiprocessing literature to exploit compile-time scheduling [KONSTANTINIDES90], [YEN95]. Such efforts typically assume a fixed or highly predictable communication structure among known computational elements. Such restrictions are well modeled by *dataflow* computational models, in which a computation is described by the flow of tokens along a graph of computational operators, without explicit control structure. The SCORE model is essentially a dataflow on CP-sized macro-operators, each of which clusters traditional dataflow operators (specifically, integer-controlled dataflow operators in our restricted SCORE model).

Synchronous Dataflow (SDF) is a dataflow computational model in which the number of tokens consumed and produced in each firing of an operator is known at compile time. SDR is thus amenable to static scheduling with minimal runtime overhead. Although SDF is not Turing-complete due to lack of conditional control, it is sufficient for many digital signal processing tasks (i.e. FIR/IIR filtering). A theoretical framework exists for statically scheduling SDF graphs on uniprocessors [BHATTACHARYYA96], which can find (or disprove the existence of) periodic firing schedules with guaranteed memory requirements and deadlock-free operation. Boolean-controlled Dataflow (BDF or Token Flow, [BUCKLEE92], [BUCKLEE93], [BUCK93]) and Integer-controlled Dataflow (IDF, [BUCK94]) are Turing-complete extensions of SDR that add simple conditional operators. Scheduling of BD and IDF graphs on uniprocessors typically requires clustering subgraphs to run in successive phases, so as to bound memory requirements.

Scheduling dataflow graphs on parallel hardware has additional synchronization complications due to: *(i)* heterogeneous operator firing times, *(ii)* network delays, and *(iii)* clustering of operators on processors. [WILLIAMSON96] implements a mapping of SDF to VHDL for hardware generation, where the creation of arbitrary control and synchronization signals obviates the need for operator clustering. With regards to clustering on conventional multiprocessors, there has been some work in compile-time scheduling based on run-time profiles [HA97] as well as static graph analysis [BHATTACHARYYA95] [PINO95]. Fully dynamic scheduling, due to its high cost, is typically not the best solution in computational domains which have static guarantees, such as SDF. [LEE91] defines a taxonomy and discusses tradeoffs in the spectrum between fully-static and fully-dynamic dataflow scheduling.

[JONSSON96] describes a heuristic, on-line, SDF scheduling algorithm for idealized message-passing multiprocessors similar in some respects to the SCORE-based reconfigurable processor. The algorithm exploits pipelining by scheduling “linear clusters” of dataflow operators. Each node in such a cluster has exactly one dataflow successor in the cluster, so the nodes form a pipeline for tokens. The study reports 80-90% utilization in the used processors for several feed-forward applications and 10% utilization for a feedback-constrained application. The study does not discuss memory constraints for data streams entering or leaving clusters, so it is possible that the utilization reported is high due to large or infinite memory assumptions.

2 Methodology & Implementation

2.1 Overview of System Flow

Figure 9 shows the overall system flow of the current SCORE implementation, from operator instantiation to actual execution of the operator on the simulator. This figure represents the simplest situation: one single-threaded user application instantiating a single-operator design. The current system simulates the reconfigurable array fabric.

The basic structure of a SCORE application is shown in Figure 9. At the start of the program, *score_init()* is called to initialize any SCORE-related variables as well as establish an interface with the runtime system. Any streams needed for passing data to or receiving data from the operator are instantiated so they can be passed to the runtime during operator instantiation. Then, the operators making up the design are instantiated. An IPC[3] message is sent to the SCORE runtime, where it is received by the IPC thread. The application then performs any necessary computations and feeds data tokens to the operators for processing (via stream writes) and receives result tokens (via stream reads). It is expected that the bulk of the time will be spent in this step. Finally, once the application has completed its task, final cleanup is performed by calling *score_exit()*.

The instantiated SCORE *operators* are abstract representations of algorithmic data transformations. Operators do not necessarily correspond to compute pages. It is the responsibility of the page partitioner to either decompose large operators or merge small operators into fixed size pages. However, because the current SCORE implementation lacks an automatic page partitioner, the page partitioning is performed by hand. For simplicity of hand partitioning, we have assumed that each operator is a self-contained composition of compute pages. The current scheduler data structures reflect this simplification (see Section 6.1.2). Future implementations will reflect the true nature of operators once a partitioner exists.

The SCORE runtime is implemented as a user-level application consisting of 3 threads (see Figure 9): IPC, scheduler, and simulator. The role of each thread is:

?? **IPC**: to receive operator instantiation requests from the IPC message queues, retrieve operator instances from persistent storage, instantiate the operator, and perform any necessary preprocessing before handing off the operators to the scheduler thread.

?? **Scheduler**: to accept preprocessed operators from the IPC thread, decide which parts of operators should be scheduled onto the hardware, and issue reconfiguration commands to configure CPs and CMBs.

?? **Simulator**: to simulate the functionality of the reconfigurable array, accept reconfiguration commands from the scheduler thread and simulate the behavior of the scheduled pages and segments.

When an operator is instantiated from the SCORE application an IPC message is sent to the runtime containing a fully-resolved filename for the location of operator data along with instantiation parameters, such as operator bit width. The IPC thread receives the instantiation message and retrieves the file containing the operator data from the filesystem. The operator is instantiated with the provided instantiation parameters. The thread initializes data structures as well as performs clustering and optimizations before entering the operator into the shared scheduler data structures. If the scheduler thread is currently idle, a "reawaken" signal is sent to begin scheduling.

The scheduler is invoked at fixed timeslice intervals. Once invoked, the scheduler examines the state of the array as well as its waiting lists to determine which pages and segments to schedule next on the array. Then the scheduler issues reconfiguration commands in the form of hardware API calls. The simulator provides a cycle-by-cycle simulation of the scheduled pages and segments. The hardware API serves as an abstraction layer so that when the simulator is replaced with real hardware, the scheduler will not need to be altered.

Scheduler and array execution occur concurrently. At the beginning of the timeslice, the scheduler reads status from the array and then allows the array to continue executing. When the scheduler is ready to issue reconfiguration commands it stops activity on the array (see Figure 10). The advantage of this technique is that the array does not sit idle while the scheduler makes its decision. However, the consequence is that the scheduler may be working on stale array status. Minimizing scheduler decision time can reduce the staleness of the status.

[3] **I**nter-**P**rocess **C**ommunication

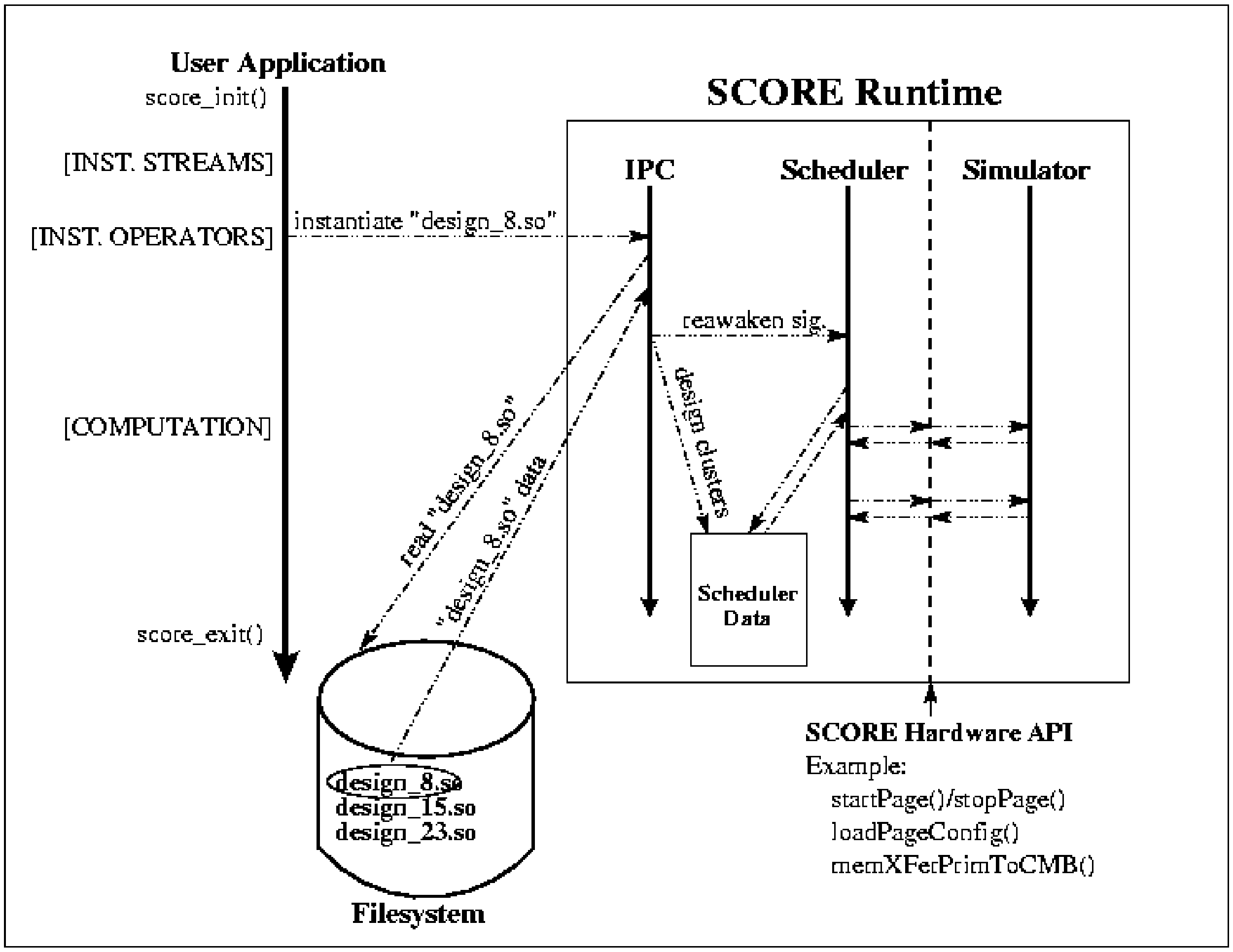


Figure 9 - High-level SCORE System Flow & Interaction

2.2 Scheduling Scenarios

Before delving into the details of the scheduler flow and algorithm, it is beneficial to recognize common scheduling scenarios and the role of the scheduler in each situation. There are three key scenarios that illustrate the types of scheduling decisions that need to be made by the SCORE scheduler. In general, most SCORE execution can be classified as one of the following models: a small design that fits completely within a large array, a large design that does not completely fit within a small array, and a design that is bufferlocked (see Section 2.4.3) as a result of finite physical resources. This section explains the each of the basic scenarios as well as the role of the scheduler in each scenario.

2.2.1 Small Design, Large Array

The simplest scenario to imagine is when the user instantiates a design fitting completely within the physical array (for an example, see Figure 6). In this case, the scheduler performs three major tasks:

?? Determine physical placement of the pages and segments on the array (Section 2.3.2.8),

?? initiate array configuration (Section 2.3.2.9), and

?? wait for pages and segments completion before removing them from the array (Section 2.3.2.3).

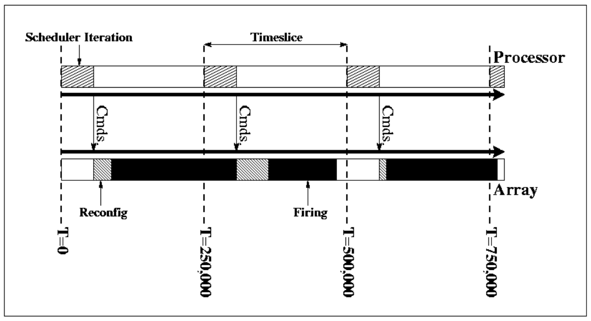


Figure 10 - Concurrent Operation of Processor & Array

The reconfigurable array is not time multiplexed and the scheduler makes no meaningful scheduling decisions.

2.2.2 Large Design, Small Array

As programmers develop more complex applications in SCORE, the more common scenario will consist of a design so large that it no longer fits in a single array (for an example, see Figure 7). In this case, the scheduler must time multiplex the limited physical array among the pages and segments of the logical design. Changes in the schedule can be made at fixed timeslices or more frequently when the status of the array changes, such as when a page stalls on lack of input.

In the current implementation, the scheduler uses fixed timeslices (see Section 2.3.2). At each timeslice, the effectiveness of the current design mapping is determined by sampling the array status. If the current mapping is determined ineffective (perhaps due to insufficient input data), the scheduler decides on the next set of pages and segments to map. Then, these nodes are placed, loaded, and run in a manner similar to Section 2.2.1.

The goal of the scheduler in this case is to rapidly determine the best set of pages and segments to schedule at each timeslice. This task can be improved by preprocessing the user design. Pages and segments working closely with one another, such as feedback loops, can be marked together in a cluster (see Section 2.3.1.1). By performing this preprocessing once at the beginning, effective scheduling decisions can be made more quickly at each timeslice.

2.2.3 Bufferlocked Design

Finally, sometimes logically correct designs may exhibit deadlock when mapped onto physical hardware. The cause of this deadlock is the limited size of the physical stream buffers versus the abstraction of infinite stream depth in the SCORE model. This can occur with designs that fit completely within the physical array as well as designs which must be multiplexed. When this occurs, the design is said to be bufferlocked.

Since it is the responsibility of the scheduler to provide the abstraction of infinite hardware, the scheduler must detect and resolve this condition when it occurs. The simplest way to do this is to search for bufferlock on every timeslice. A more efficient method is to wait for the design to deadlock and then run the bufferlock detection routine. Once bufferlock is found, the bufferlock cycle is broken by artificially increasing the stream buffer depth by inserting CMBs to serve as FIFOs. Section 2.4.3 goes into more depth on the exact method used by the current scheduler implementation.

2.3 Scheduler Flow

This section explains the actual execution flow of the implemented scheduler. There are two types of flow in the SCORE scheduler: operator instantiation flow and timeslice iteration flow. Operator instantiation flow includes the sequence of steps performed by the scheduler every time an operator is instantiated. For most applications, this flow is experienced only once at the beginning of the run. Timeslice iteration flow describes the sequence of steps performed by the scheduler at every timeslice and is incurred multiple times in a run, depending on the application.

2.3.1 Operator Instantiation

Operator instantiation in the runtime system is handled by the IPC thread. Figure 9, shows the sequence of actions once an operator is instantiated by the user application. An IPC message containing the fully-resolved filename for the operator data is sent to the IPC thread within the runtime. The operator data is retrieved from the filesystem and the operator is instantiated with the given parameters. The result of instantiation is passed to the *addOperator()* method responsible for performing checks on the operator and preparing it to be scheduled.

2.3.1.1 Adding a new operator

Figure 11 shows the execution flow of *addOperator()*. Once *addOperator()* receives the instantiation of an operator, it is responsible for initializing the operator variables and preparing the pages and segments in the operator to be scheduled by the scheduler thread.

Acquiring and releasing the lock on scheduler data structures prevents corruption of scheduler data structures from simultaneous changes by the IPC and scheduler threads. The major operations performed include running the SCC[4] graph algorithm [CORMEN96] for cluster formation, checking physical constraints on the clusters, initialization and bookkeeping, and insertion of the clusters into the scheduler’s waiting list.

The SCC graph algorithm takes a given directed dataflow graph and decomposes it into its strongly connected components. Strongly connected components of a graph consist of nodes which “are mutually reachable”. “Mutually reachable” means that starting from any node in a strongly connected component we

[4] SCC: Strongly-Connected-Components

are able to reach any other node in the component by traversing edges. Figure 12 shows an example of a dataflow that has its strongly connected components marked. This property corresponds to feedback loops among pages and segments in an operator. We attempt to identify feedback loops in the dataflow because the nodes within a feedback loop must be handled in a special manner. If nodes in a feedback loop are allowed to be non-coresident, then the array will experience configuration thrashing. On each timeslice, the resident portion of the feedback loop will be able to process only a few tokens before it is starved for data provided by the non-resident portion. To prevent this situation from happening, once the system identifies the nodes of a feedback loop, those nodes are placed in a cluster which guarantees they will be scheduled on the array atomically.

Once the operator is partitioned into clusters, *addOperator()* checks each cluster's physical resources requirement. One of the requirements of a cluster is that is should be able to be scheduled on the array by itself to prevent schedule starvation (see Section 6.1). To guarantee this requirement, *addOperator()* examines each newly formed cluster and counts the number of pages that exist in the cluster. This number must be less than or equal to the number of physical CPs in the array. Next, it counts the number of segments plus the number of cluster IO streams. This number must be less than or equal to the number of physical CMBs in the array. The reason for counting the number of cluster IO streams is to determine the number of stitch buffers that could potentially be required. A cluster that does not meet these requirements is then decomposed into a smaller cluster with, hopefully, fewer requirements. In the current implementation, invalid clusters are decomposed into single-node clusters (i.e. only one page or segment in each cluster). Any remaining clusters that still do not pass the test will cause the entire operator to be rejected from the runtime system.

If all clusters pass the check, *addOperator()* initializes the variables for the operator, pages, segments, and clusters. Then, some bookkeeping operations are done to synchronize the state of the system with the addition of the operator. These bookkeeping operations include: adding the operator to the parent process object, maintaining the processor-array IO stream list, and adjusting the "frontier" scheduling head cluster list (see Section 2.4.1 for more explanation on the head cluster list).

Finally, just before *addOperator()* finishes, the newly formed clusters are added to the scheduler's waiting cluster list. There they await scheduling during the next timeslice.

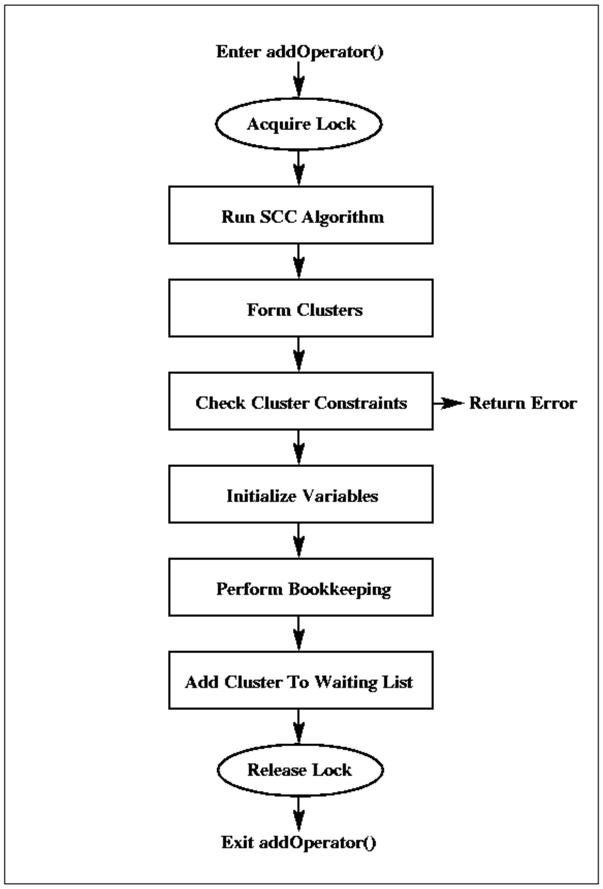


Figure 11 - Execution Flow of addOperator()

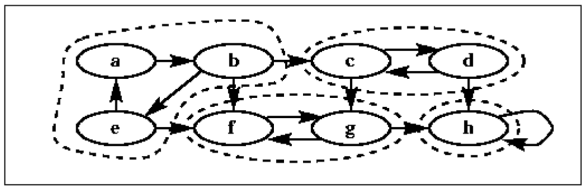


Figure 12 - Dataflow Graph with Strongly Connected Components Marked[5]

2.3.2 Timeslice Iteration

Scheduling decisions are made at timeslice intervals. At predetermined times defined by the timeslice interval, the scheduler thread is woken up, examines the current state of the reconfigurable array and decides which portion of the dataflow to schedule on to the array and which to swap out. *doSchedule()* is the method called at each timeslice and it in turn calls several other methods to perform status gathering and scheduling (see Figure 13).

[5] Adapted from Figure 23.9 on page 489 in [CORMEN96].

The entire process consists of several stages. At the beginning of the timeslice, the status of the array is read. The array is allowed to continue executing while the scheduler makes its decision to hide the overhead of scheduling. Using the “frontier” scheduling heuristic (see Section 2.4.1), clusters are marked to be removed or scheduled. Placement is performed to determine where the scheduled pages and segments will reside on the array. Finally, array execution is halted and reconfiguration commands issued to dump and load the appropriate pages and segments. The array is restarted and final cleanup is performed. Like *addOperator()*, *doSchedule()* acquires and releases the lock on scheduler data structures to prevent the corruption of the data structures from simultaneous changes by the IPC and scheduler threads.

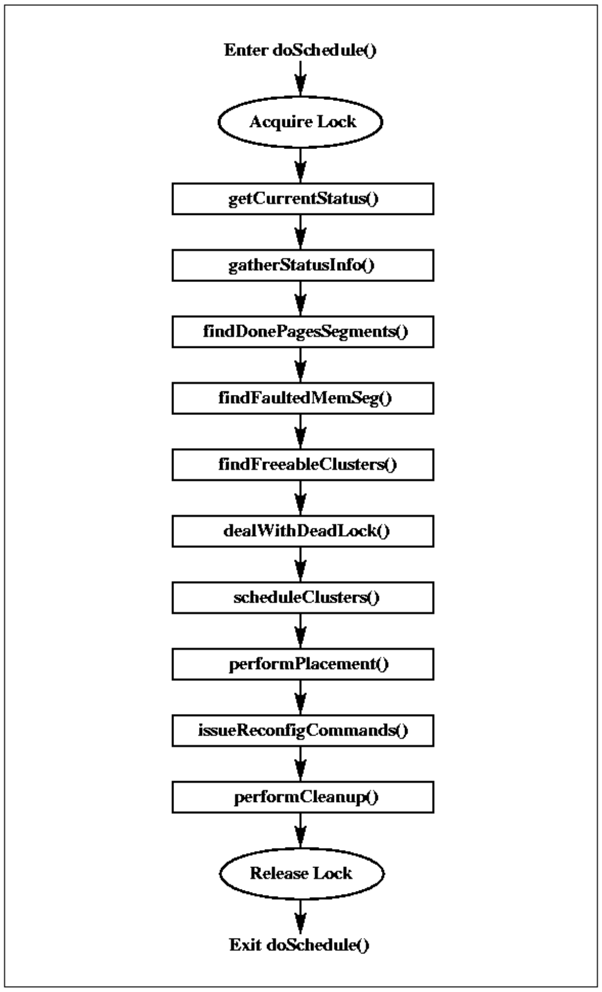


Figure 13 - Execution Flow of doSchedule()

2.3.2.1 Retrieving physical array status

After *doSchedule()* acquires the scheduler data lock, the scheduler reads the status from the physical array. The status includes:

?? Which pages and segments are stalled on input or output streams along with the number of cycles each node has been stalled and which streams are causing the stalls.

?? Which pages and segments have finished executing and signaled done.

?? The stream consumption and production rates for each input and output stream.

?? For each page, the state its state machine has reached.

?? For segments, the memory address causing the address fault if the segment has experienced a fault.

?? The number of tokens left unprocessed in the stream inputs FIFOs.

The *getCurrentStatus()* method reads the array status through the hardware API *getArrayStatus()* call (see Section 2.4.4). The array is allowed to continue executing while the scheduler interprets these results. The raw status is passed to the *gatherStatusInfo()* stage for processing.

2.3.2.2 Convert and process array status

The purpose of the *gatherStatusInfo()* scheduler stage is to convert the raw status information from the hardware into usable information for the scheduler. Raw status is returned in a compact array with each element in the array corresponding to a physical CP or CMB. The lack of correlation between the dataflow representation and the physical status array makes it more difficult to utilize the information during scheduling.

It is the responsibility of *gatherStatusInfo()* to traverse the physical status array, look up the mapping to the virtual page or segment in the dataflow graph and transfer the status to the graph node.

2.3.2.3 Detect done pages and segments

As pages and segments complete their execution, a built-in mechanism allows these nodes to signal to the runtime system that they can be removed. This mechanism is the done signal which is an out-of-band (i.e. independent of the stream communication) signal to the processor. This signal is part of the status returned by the hardware to the scheduler.

The scheduler subdivides done nodes into two types: explicit done nodes and implicit done nodes. Once the scheduler receives a done signal from a page or segment, that node is marked as a done node and scheduled for removal from the array. When a node is marked in this manner, it is referred to as an explicit done node.

However, signaling done is not the only way for a node to be marked done. A node can also be marked done if it no longer has a reason to exist in the dataflow graph. For a page, this means that all consumers of its outputs have signaled done. Therefore this page can no longer affect the final result of the operator. For a segment, not only must all of the consumers of its output be done, it must also guarantee that it can no longer affect the data block associated with the segment. This is simple if the segment is read-only. However if a segment is write-only or read-write, its input nodes must also be done.

A graph search is performed, starting from the explicit done nodes, to find the implicit done nodes. As implicit done nodes are discovered, they are added to the search list. A node marked done because it is logically useless in the dataflow graph is referred to as an implicit done node.

Figure 14 shows an example of how nodes would be marked explicit and implicit by the scheduler. All nodes in the figure are pages. Assuming node C signaled done, node C would be marked as an explicit done node. Since it is the only consumer of output tokens from node B, node B is logically useless and marked as an implicit done node. In turn, node A becomes logically useless and is also marked as an implicit done node. Node D is not marked as an implicit done node because its output tokens are also consumed by node E which is not done. Finally, node F does not have its output stream tokens consumed by node C and is therefore unaffected by node C signaling done.

All done nodes (explicit or implicit) are placed on a done node list which is passed on to later scheduler stages. The nodes are removed from their parent clusters, operators, and processes. If it is currently scheduled, its position on the array is cleared to make room for non-done nodes. Its associated data structures (i.e. C++ object representation in scheduler memory) are cleaned up during the *performCleanup()* stage (see Section 2.3.2.10).

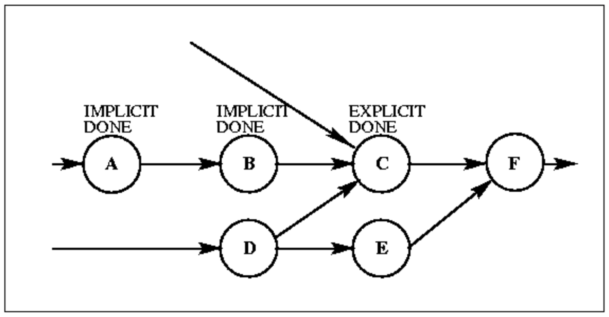


Figure 14 - Example of Explicit and Implicit Done Nodes

2.3.2.4 Detect address-faulted memory segments

The SCORE compute model permits user-instantiated segments to be arbitrarily large. Physically, the scheduler realizes this abstraction by loading a fixed-size block of data into a CMB and appropriately setting the base and bound registers within the CMB to accomplish address translation (similar to paged virtual memory systems). The base and bound registers also serve to notify the scheduler of memory accesses made outside of the currently loaded block. In this case, an address fault signal is sent to the processor and the address causing the fault is recorded.

In *findFaultedMemSeg()*, the scheduler examines the physical array status to determine which segments have experienced address faults. It must determine if: (i) this is a genuine address fault caused by memory virtualization, or (ii) this is a segmentation fault caused by the operator trying to access memory outside of the range defined for that segment. If it is a genuine address fault, the scheduler determines the next block of data to load. If it is a segmentation fault, the operator is terminated with an error.

In the future, it is likely that this functionality will be moved to a separate memory management thread. This would enable address faults to be serviced without the overhead and latency of a full scheduler iteration. In this case, it would be beneficial to have a separate path available to transfer memory between CMBs and main memory to avoid interrupting running nodes on the array when performing memory management.

2.3.2.5 Determine and mark freeable clusters

When pages and segments become stalled, the scheduler must decide which nodes to remove from the array to make room for waiting pages and segments. This task is performed by the *findFreeableClusters()* stage. In this stage, the scheduler examines the list of currently scheduled clusters to determine which clusters should be removed from the array. The decision is based on how many nodes in each cluster are stalled and cannot make forward progress. There are two issues to consider when making this decision:

?? How does the scheduler determine that a node can no longer make forward progress?

?? How does the scheduler determine when to remove a cluster if some of the cluster nodes are still able to make forward progress?

Simply looking at which nodes are currently stalled is insufficient. In the course of execution, most nodes will be stalled on inputs or outputs some time due to the latency inherent in passing tokens through the network. Instead, the scheduler applies a heuristic based on the number of cycles a node has been stalled. The notion of stall threshold is defined as the maximum number of cycles a node can be stalled in a timeslice before it is no longer considered able to make reasonable forward progress. Currently, the stall threshold is set to be one-half of the timeslice width in cycles. Nodes considered unable to make reasonable forward progress are marked freeable nodes.

Once the freeable nodes have been identified, the scheduler needs to decide which clusters should be removed. This is done by traversing the resident cluster list and examining the status of the cluster nodes. If all of the nodes of a cluster are marked freeable, the parent cluster is marked freeable.

However, the decision is more difficult if only some of the nodes are marked freeable. The reason is because the semantics of clusters requires that all of its nodes either be atomically scheduled or removed from the array. This would mean potentially preempting a node which can still make reasonable forward progress. This issue is resolved with another heuristic. The notion of a cluster freeable ratio is defined as the ratio of nodes in a cluster that must be marked freeable before the cluster itself is marked freeable. Currently, the cluster freeable ratio is set to 0.5, meaning that one-half of the nodes in a cluster must be marked freeable before the cluster is marked freeable. All clusters marked freeable are placed on a freeeable cluster list which is passed on to later scheduler stages[6].

[6] It should be noted that the classification of freeable clusters is only a recommendation. There is no guarantee that a freeable cluster will be removed during this timeslice or clusters not marked as freeable will not be removed.

2.3.2.6 Detect and resolve runtime deadlock

During the course of execution, it is possible for a user’s design to experience deadlock. Deadlock may result from an inherent flaw in the dataflow or introduced by the scheduler due to physical stream constraints. To determine the cause of the deadlock, deadlock detection must be performed on the design. The method of deadlock detection and resolution is described in Section 2.4.3.

Deadlock detection potentially could be performed on every timeslice to immediately detect deadlocked designs. However, the heuristic used to perform detection is expensive. In most timeslices, there will be no deadlock to detect and the detection overhead will be wasted. Therefore, the scheduler attempts to wait until a design is actually deadlocked before deadlock detection is performed.

The method used by the scheduler is simple. On every timeslice, the scheduler looks at the status for each resident page and segment to determine if it has consumed any inputs or produced any outputs. If a node has done neither, it will be marked as non-firing and the count of non-firing nodes for that user process is incremented. Likewise, if a node does consume inputs or produce outputs during a timeslice and it was previously marked non-firing, the mark is reset and the process count of non-firing nodes is decremented. Once a process’s non-firing node count equals the number nodes in the process, it is subjected to deadlock detection.

It is still possible for deadlock detection to be prematurely run on a process. In this case, all of the nodes are reset to firing status and the non-firing node count is reset. This situation could occur for various reasons, including: the application for this design has not injected tokens or the non-firing status marks are stale or inaccurate (a node may be firing without consuming inputs or producing outputs).

2.3.2.7 Dynamically schedule clusters

The *scheduleClusters()* stage is the heart of the SCORE scheduler. It is responsible for determining which clusters are actually scheduled and removed during the current timeslice. To perform this task, it utilizes the done node list and freeable cluster list to understand the current state of the array. It then proceeds using trial-based scheduling to determine which clusters should be removed and scheduled.

The trial-based scheduling method works in the following manner: for each scheduling trial, the scheduler performs a scheduling action, such as adding or removing a cluster. After each scheduling action, the scheduler calculates the number of physical CPs and CMBs that would be free (unoccupied) if no more scheduling actions are performed.

The number of free physical CPs is required to remain non-negative. The scheduling trials are halted if the speculated number of free physical CPs goes to zero or a negative number. However, the speculated number of free physical CMBs is allowed to drop below zero. The trials are ended when either the speculated number of free CPs reaches zero (or would have become negative) or no more clusters remain waiting to be scheduled. At this point, the scheduler backs up to the last trial where the number of free CPs and free CMBs are both non-negative. The scheduler must maintain enough intermediate information to perform the trial rollback.

Figure 15 shows an example of trial-based scheduling. Initially, there are no clusters resident on the 4-CP/4-CMB array. Each circle represents a single-page cluster. Therefore, at the beginning of the trials, free CPs equals 4 and free CMBs equals 4.

The scheduler selects cluster A for scheduling. Immediately, we notice that the number of free CMBs drops to 0. The reason is because of the speculative addition of stitch buffers. Stitch buffers are segments added to serve as a token source or token sink when only one end of a stream is resident on the array. Stitch buffers are realized using FIFO segments. If it is only sourcing tokens, it is placed in read-only mode. If it is only sinking tokens, it is placed in write-only mode. If both the source and sink of the stream become resident but tokens still remain in the stitch buffer, it is placed in read-write mode.

In successive trials the number of free CPs continues to decrease by 1 while the number of free CMBs fluctuates with the choice of clusters scheduled. The trials are halted when all of the free CPs have been exhausted. At this point, because the number of free CMBs is non-negative, no roll back of trials is needed.

In the current implementation of the scheduler, scheduling trials progress in the following order:

?? Done nodes are removed from the array.

?? If there are clusters on the waiting list, clusters on the freeable cluster list are speculatively removed.

?? Clusters are repeatedly added to the array speculatively using the “frontier” scheduling heuristic (see Section 2.4.1) until no more free CPs remain[7] or no more clusters remain unscheduled.

Once the scheduling trials have been completed and the last valid trial has been selected, the scheduler updates the internal cluster lists (i.e. waiting cluster list, resident cluster list) and appropriately marks each cluster, page, and segment with its residency status. Stitch buffers are inserted into the dataflow graph where appropriate. The outputs generated by the *scheduleClusters()* stage are lists of scheduled pages and segments as well as removed pages and segments.

2.3.2.8 Determine page and segment placement

After clusters have been scheduled and removed, *performPlacement()* is responsible for assigning the exact physical locations for the cluster nodes. In this stage, memory management is also performed; the memory blocks where page configuration and state as well as segment data will exist are assigned. These assignments are passed to the *issueReconfigCommands()*. The inputs into the *performPlacement()* stage are the lists of scheduled and removed pages and segments from the previous stage, *scheduleClusters()*.

Before assigning locations to the newly scheduled nodes, the scheduler’s array view is updated to reflect removed pages and segments. The memory blocks containing the configuration, state, and data of the removed nodes are marked as cached information (except for done nodes, whose associated memory blocks are marked as empty).

The scheduler then examines the lists of scheduled pages and segments. To maximize the benefits from caching configurations, state, and data in array CMBs, the scheduler tries to lock down already cached information. The associated memory block is marked as used and cannot be evicted.

After cached information is identified and properly protected from eviction, the scheduled pages and segments are placed in free CP and CMB locations. Currently, there are no restrictions as to where individual pages can be placed. Locations for pages are randomly selected from a list of free CPs. However, there are restrictions associated with where segments can be placed. The CMB where a segment is placed must also contain a contiguous memory block large

[7] As part of the “frontier” scheduling heuristic, clusters originally on the freeable cluster list may be rescheduled on the array if enough space exists.

enough to hold the segment data. Luckily, the current memory management scheme guarantees this to be true for each free CMB location (more information about the memory management scheme is given below).

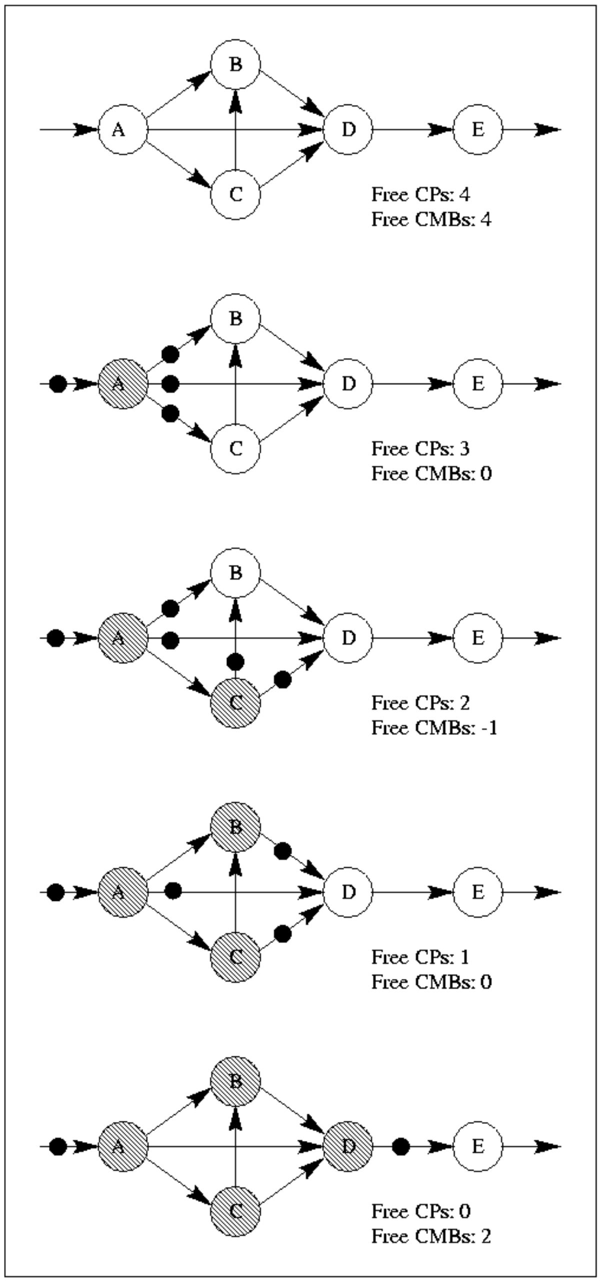


Figure 15 - Example of trial-based scheduling. Shaded indicates speculatively scheduled and dots indicate speculative stitch buffers.

Once all of the pages and segments have been placed into free CPs and CMBs, any remaining pages that does not have their configuration and state cached on the array will have a free memory block allocated. No further memory allocation is needed for segments, because the CMB location dictates the location of the segment data. For pages, the location of its configuration and state cache is unimportant[8] and therefore can be allocated in any free memory block. At this time, *performPlacement()* will arrange for faulted memory segments to have the next data section loaded from primary memory.

The memory management scheme used to allocate memory blocks is the pseudo-“buddy” system memory management. It is a derivative of the “buddy” system [KNUTH73]. The pseudo-“buddy” system attempts to minimize internal fragmentation while making it easy to find free memory blocks of the necessary size. A more in-depth discussion of this memory management system is in Section 2.4.2.

The result of the *performPlacement()* stage is an updated *arrayCP* and *arrayCMB* physical array view. For each element of these arrays, the *scheduled* entry will be appropriately set to the page or segment scheduled in that location. It will be the responsibility of the *issueReconfigCommands()* stage to update the *active* entry once reconfiguration is completed. In addition the segment block table for each CMB is updated with the most recent memory block allocations.

2.3.2.9 Issue reconfiguration commands

By the time the *issueReconfigCommands()* stage is reached, the scheduling decisions for the current timeslice have been made. It is the responsibility of this stage to issue the reconfiguration commands so that the physical array reflects the scheduling decisions. This is done by stopping execution of the array, issuing reconfiguration commands via the hardware API (see Section 2.4.4), and then restarting execution of the array. Afterwards, *arrayCP* and *arrayCMB* are updated to reflect the reconfiguration (i.e. the *scheduled* entry is copied to the *active* entry).

The first action performed by the stage is to stop all execution of CPs and CMBs. The decision to stop the entire array stems from the desire to simplify the scheduler. This eliminates the need to determine the portion of the array that would become affected by reconfiguration. Given an array network structure where this determination is simple, performing partial array halting may be considered.

After the array has been halted, reconfiguration commands are issued to load/dump CMB memory blocks, reconfigure CPs/CMBs, and connect

[8] It is assumed that the network allow reconfiguration of CPs from any CMB on the array.

input/output streams. The scheduler attempts to issue multiple reconfiguration commands in parallel, as long as there are no resource conflicts. The *batchCommandBegin()* and *batchCommandEnd()* pair are used to designate a parallel command set. The packing of parallel reconfiguration commands is done in a manner similar to scoreboarding [HENNESSY90]. As commands for reconfiguration are issued, the CPs and CMBs involved are marked occupied. Future commands are blocked from being packed in the batch if they attempt to utilize a busy resource (i.e. CMB or CP). The busy marks are cleared at the completion of a batch command.

Once the array has been reconfigured, the entire array is restarted with the new configuration. The scheduler will not examine the status again until the next timeslice.

2.3.2.10 Perform cleanup operations

The final stage of the scheduler is responsible for performing garbage collection actions resulting from nodes signaling done as well as stitch buffers becoming empty. For each done node, *performCleanup()* looks at the parent cluster, operator and process objects. If the node is the last remaining member, the appropriate parent object will be deleted. For done segments, access to the memory region is returned to the user process.

In addition to done nodes, the stage also deletes empty stitch buffer objects no longer needed in the dataflow. This is only performed for stitch buffers which are removed from the physical array. This guarantees that the state and address pointers for the stitch buffer are in a known and stable state.

This stage is not in the critical path of the scheduler iteration. While currently inline with the other stages in *doSchedule()*, it may be possible to create a separate thread of execution responsible for cleanup. In that case, adequate protection of the scheduler data via locks needs to be implemented in *performCleanup()* to guarantee nodes are not being deleted at the same time they are being accessed.

2.4 Scheduler Implementation Highlights

The SCORE scheduler contains several implementation details which merit more in-depth discussion. In the following sections, implementation highlights of key areas are described, including: the scheduling heuristic, memory management scheme, deadlock detection, and simulator interfacing.

2.4.1 “Frontier” Scheduling Heuristic

The SCORE scheduler implements a specialized version of priority-list scheduling. With traditional priority-list scheduling, it is often the case that all waiting tasks have equal opportunity to become read to run. As a result, all waiting tasks must be examined when making a scheduling decision.

However, given the large overhead of CP reconfiguration, SCORE lends itself to applications with largely feed forward dataflows. Feed forward dataflows allow the compute pages in the design to execute for longer periods of time, thereby amortizing reconfiguration overhead. Another consequence of feed forward dataflows is that ready to run waiting tasks are largely isolated to the periphery of the scheduled dataflow. The current scheduler implementation attempts to take advantage of this fact to optimize scheduling time.

The “frontier” scheduling heuristic separates the traditional priority waiting list into a prioritized “frontier” cluster list and a waiting cluster list. During a normal scheduling iteration, the next cluster to schedule is selected from the “frontier” list. The cluster’s successor are removed from the waiting list and placed on the “frontier” list. Figure 16 shows an example dataflow with various elements of “frontier” scheduling highlighted. In this example, cluster A is resident on the array, therefore placing cluster D on the “frontier” list. Cluster B is on the “frontier” list because it was on the head list (the head cluster list will be explained below). This leaves clusters C and E on the waiting cluster list.

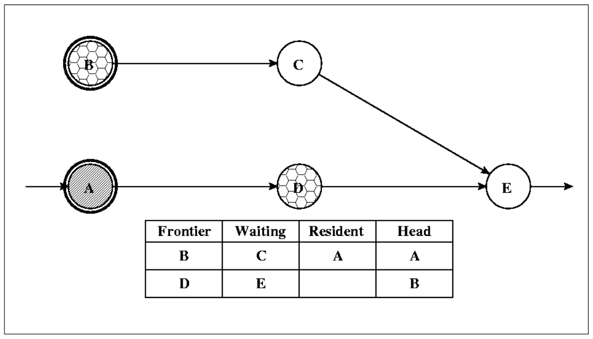
There are some special cases that need to be considered for the heuristic to function correctly:

?? How does the scheduling heuristic begin scheduling when the runtime is initialized?

?? How does the scheduling heuristic continue scheduling once the end of the dataflow is reached?

?? How does the scheduling heuristic handle inter-cluster feedback loops[9]?

When the runtime is started the “frontier” cluster list is initialized to be empty. There needs to be a way to seed the scheduling heuristic. By maintaining a head cluster list from which the “frontier” cluster list is loaded when empty, this special case is resolved. Membership in the head list is defined to guarantee starvation will not occur.



Frontier	Waiting	Resident	Head
B	C	A	A
D	E		B

Figure 16 - Example of Various Elements of "Frontier" Scheduling

A cluster is placed on the head cluster list if:

?? It has no input streams, or

?? It has an input stream originating from the processor, or

?? It has an input stream whose producer has signaled done.

Figure 19 shows that clusters A and B satisfy the above criteria for being on the head list.

Once the heuristic reaches the end of the dataflow, there needs to be a way to continue the scheduling heuristic. This case is also resolved with the head list. When the heuristic reaches the end of the dataflow, eventually the frontier list will become empty (unless there are inter-cluster feedback loops, which is addressed below). At this point, the “frontier” list is loaded with the head list and the cycle is started again.

Unfortunately, the heuristic has an undesirable behavior when presented with a design containing inter-cluster feedback loops[9]. The potential problem is that inter-cluster feedback loops will artificially keep the “frontier” list filled, preventing the heuristic cycle from restarting at the head list. The result is an oscillating state as shown in Figure 17. This figure shows that, given an array capable of holding either cluster R or cluster S but not both, the system will oscillate between cluster R resident and cluster S resident; cluster Q will never be scheduled again. This must be handled to avoid cluster starvation during scheduling. The current implementation avoids this pathological case by keeping track of the current “traversal” of the heuristic and allowing each cluster to be scheduled only once per “traversal”. The traversal count is incremented every time the “frontier” cluster list becomes empty and is reloaded from the head list. This avoids starvation in all dataflow configurations assuming the head list is properly maintained.

[9] Inter-page feedback loops are okay as long as they are completely contained within a cluster. Inter-cluster feedback loops can result from two situations: (i) insufficient physical resources forcing a cluster to be decomposed, exposing the feedback loop; (ii) a feedback loop spanning multiple operators (i.e. C++ compositional operators).

2.4.2 Pseudo-“Buddy” System Memory Management

One of the ways to extract the necessary performance from the reconfigurable array is by caching CP configuration/state/input FIFOs as well as CMB data/input FIFOs in array CMBs. This allows reconfiguration to, theoretically, be done completely in parallel to/from array memory as opposed to serially across the CPU/array interface. However, intelligent management of CMB memory space is necessary to avoid pathologically bad caching that can lead to on-chip serialization[10].

The ideal memory management strategy is to allocate variable-size segments tailored to the size of what needs to be cached. This avoids the internal fragmentation associated with fixed-size pages. However, as caching activity increases, external fragmentation increases. Also, the cache management overhead is complicated to quickly find a best fit segment.

On the other extreme are a fixed-size pages. Fixed-size pages avoid the external fragmentation associated with variable-size segments. However, the sizes of the configuration, state, and input FIFOs and the memory segment data differ significantly. This means either a large page size or multiple small-pages for caching. There is an incentive for keeping memory segment data contiguous in CMBs; it is assumed that there is only one set of address translation registers in each CMB. Unfortunately, a single large page size incurs significant internal fragmentation for configuration, state and input FIFOs.

The compromise used by the scheduler is the pseudo-“buddy” system. It is derived from the “buddy” system [KNUTH73] [KNOWLTON65]. The “buddy” system is a memory management scheme that attempts to speed up the search for appropriately sized free blocks (a problem with variable-sized segments) while

[10] One example of on-chip serialization can be shown through the example of caching all compute pages in one CMB.

trying to minimize the amount of internal fragmentation (a problem with fix-sized pages). By successively dividing the memory space into halves, the "buddy" system is able to more tightly fit a free block to the data. If a large free block needs to be recovered, the smaller blocks to free are easily determined.

However, the runtime does not need the full granularity of the "buddy" system. There are only two types of blocks that need to be cached in the CMBs: compute page cache and memory segment cache. The compute page cache consists of configuration, state and input FIFOs. The memory segment cache consists of data and input FIFOs. There is no advantage to separating the elements associated with the same compute page or memory segment (i.e. a single compute page cannot load both configuration and state in parallel). Memory segment cache blocks will tend to be larger than compute page cache blocks due to segment data. Therefore, instead of dividing the memory space into successive halves, the pseudo-"buddy" memory system maintains only two levels of granularity: LEVEL0 blocks (sized for maximum memory segment cache blocks) and LEVEL1 blocks (sized for maximum compute page cache blocks). (See Figure 18)

The memory space is initially divided into full LEVEL0 blocks. Any leftover space is marked as "cruft" and is unused by memory segment cache blocks. Each LEVEL0 block can be further subdivided into LEVEL1 blocks with unused space marked as LEVEL1 "cruft". In Figure 18 ("Possible Allocations"), there is an example of how an array CMB is allocated at either LEVEL0 and LEVEL1.

Each LEVEL0 or LEVEL1 block can be in one of four states: (i) free, (ii) unavailable, (iii) used, or (iv) cached. All LEVEL0 blocks are initially marked free with the corresponding LEVEL1 blocks marked unavailable. Any remaining LEVEL1 blocks are marked free.

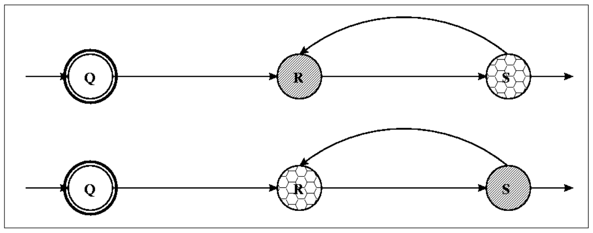


Figure 17 - Example of Inter-Cluster Feedback Loop

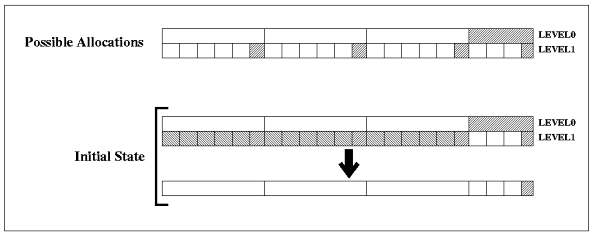


Figure 18 - Allocation of CMB Memory Blocks in Pseudo-"Buddy" System

The *free* state indicates that a block is currently not occupied and may be allocated. The *unavailable* state indicates that a block is currently occupied by another level (i.e. an unavailable LEVEL0 block indicates it is currently subdivided into LEVEL1 blocks; an unavailable LEVEL1 block indicates that is currently merged into a LEVEL0 block). The *used* state indicates that a block is currently occupied and the owner (compute page or memory segment) is currently scheduled on the array; used blocks cannot be pre-empted. The *cached* state indicates that a block is currently occupied but the owner is currently not scheduled; cached blocks can be reallocated as long as dirty blocks are swapped out first.

An example of how blocks are initialized can be seen in Figure 18 ("Initial State"). Each array CMB contains a segment table object used to keep track of which LEVEL0/1 blocks are currently free, unavailable, used, or cached. In addition to the various block lists, each segment table contains a map of the locations for the blocks in the CMB.

When a memory segment is scheduled on the array, a block is allocated in the same CMB. The scheduler first looks for a free LEVEL0 block; if there are no free LEVEL0 blocks, a cached LEVEL0 block is chosen and evicted from the CMB. The runtime guarantees that there is at least one free or cached LEVEL0 block in each CMB. No LEVEL1 blocks are ever evicted when allocating LEVEL0 blocks.

When a compute page is scheduled on the array, a similar process occurs. However, the block can be allocated in any CMB on the array. It is assumed that the routing network on the array supports configuration of a CP from any arbitrary CMB. The runtime randomly picks a CMB on the array[11] and looks for a free LEVEL1 block; if there are no free LEVEL1 blocks, but there are cached LEVEL1 blocks, a cached LEVEL1 block is chosen and evicted from the CMB.

[11] By randomly choosing the CMB, the location of the compute page configurations is evenly spread out, allowing for more parallelism in reconfiguration.

However, unlike LEVEL0 blocks, a CMB is not guaranteed to have a free or cached LEVEL1 block. Therefore, the runtime continues to search for a free or cached LEVEL1 block until it succeeds or all CMBs are exhausted. In the course of allocating a free or cached LEVEL1 block, the runtime system will also attempt to subdivide free or cached LEVEL0 blocks to obtain the necessary LEVEL1 blocks.

When a compute page or memory segment is scheduled on the array, the corresponding cached block is marked used to prevent it from being evicted. Once removed, a page's or segment's corresponding cached block is marked cached, allowing it to be evicted if necessary.

The final condition to consider is when a page or segment signals done. When a memory segment is done, its LEVEL0 block is marked free and returned to the free LEVEL0 list. When a compute page is done, its LEVEL1 block is also marked free and returned to the free LEVEL1 list. However, as a policy, the pseudo-"buddy" memory system attempts to maintain a given block of memory as a larger LEVEL0 block instead of the smaller LEVEL1 blocks. Therefore, whenever a LEVEL1 block is freed, the runtime system attempts to consolidate it into a LEVEL0 block.

2.4.3 Deadlock Detection & Bufferlock Detection/Resolution

To avoid artificially deadlocking the user's design, the scheduler must occasionally perform bufferlock detection and resolution. Bufferlock is a restricted form of deadlock that is caused by the limited token buffer space in the stream links connecting pages and segments. Bufferlock can be introduced by the runtime during the mapping of the abstract dataflow, where stream buffering is assumed to be unlimited, to the physical hardware, where real limits exist.

The scheduler performs deadlock and bufferlock detection by discovering producer-consumer dependency cycles. The scheduler starts by constructing the current producer-consumer dependency graph. Figure 19 shows an example of a design dataflow and its corresponding dependency graph. The solid edges indicate the token stream connections. The dashed edges indicate the dependency edges. Each node is annotated with dots on its input/output ports: a dot on an input port indicates the node is consuming from that input stream, a dot on an output port indicates that the node is producing to that output stream. The streams are annotated with their empty or full status.

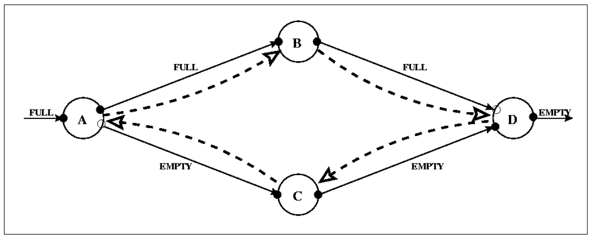


Figure 19 - Example of Dependency Cycle in Bufferlock Detection

The nodes of the dependency graph consist of the nodes of the dataflow (i.e. pages and segments). A fake processor node is inserted which consumes from all streams read by the processor and produces to all streams written to by the processor. In the dependency graph, a dependency edge exists between two nodes if

?? A node is trying to produce tokens to a full output stream (i.e. in Figure 19, A-to-B and B-to-D); in this case, the edge starts from the producer node and ends at the consumer node, or

?? A node is trying to consume tokens from an empty input stream (i.e. in Figure 19, D-to-C and C-to-A); in this case, the edge starts from the consumer node and ends at the producer node.

Once the producer-consumer dependency graph is constructed, a cycle discovery algorithm is used to return any cycles that exist, including the streams that make up each cycle. The current implementation uses a modified version of depth-first-search (DFS). The advantage of using depth-first-search is the linear complexity on the number of nodes and edges. The disadvantage of using DFS is that not all bufferlock stream cycles may be detected in one pass. It may require more than one timeslice to detect all bufferlock cycles. However, this is an acceptable compromise since a more exhaustive search would be exponential in complexity.

The result of cycle discovery is a list of cycles complete with component streams. The bufferlock detection code attempts to remove duplicate entries, as well as any cycles that cross the fake processor node[12]. Inherent deadlock cycles in the user's design can also be detected by searching for dependency cycles consisting of purely empty streams. Detected deadlock

[12] The scheduler currently does not know which streams the processor is reading from or writing to. In the future, the scheduler may be able to properly detect processor stream reads and writes.

cycles are passed to deadlock resolution. Any remaining cycles are passed to bufferlock resolution.

Bufferlock resolution attempts to insert stitch buffers to break bufferlock dependency cycles. Given a list of bufferlock cycles, it simply takes the first full stream of each cycle and inserts a stitch buffer. If the producer and consumer nodes of the selected stream are resident on the array, bufferlock resolution attempts to schedule the newly added stitch buffers onto the array.

Deadlock resolution is simple. Once an inherent deadlock is detected at runtime, the only solution available is to terminate the user design. To guard against deadlocks detected using stale data, the current implementation includes hysterisis to prevent false positives. A design must test positive for deadlock twice in a row before the design is terminated.

2.4.4 Simulator Interface

All reconfiguration is performed through a hardware API. The hardware API includes all of the commands necessary to load/dump CPs and CMBs, transfer blocks of memory, and start/stop the array. The current implementation of the API consists of the following commands:

?? *getArrayInfo()*
Returns the current hardware configuration (i.e. array size and CMB size).

?? *getArrayStatus()*
Returns the current CP/CMB status of pages/segments (i.e. stalled pages, done pages, etc.)

?? *startPage()/stopPage()*
Starts or stops the indicated CP.

?? *startSegment()/stopSegment()*
Starts or stops the indicated CMB.

?? *loadPageConfig()*
Loads a configuration for a CP from a given CMB.

?? *loadPageState()/dumpPageState()*
Loads or dumps the state for a CP from/to a given CMB.

?? *loadPageFIFO()/dumpPageFIFO()*
Loads or dumps the input FIFOs for a CP from/to a given CMB.

?? *setSegmentConfigPointers()/getSegmentPointers()*
Sets or retrieves the mode, status and address pointer registers for a CMB[13].

?? *loadSegmentFIFO()/dumpSegmentFIFO()*
Loads or dumps the state for a CMB from/to a given CMB.

?? *memXferPrimToCMB()/memXferCMBToPrim()*
Transfers a block of memory between a CMB and primary CPU memory.

?? *memXferCMBToCMB()*
Transfers a block of memory between two CMBs.

?? *connectStream()*
Connects the output port and input port of two CPs/CMBs via the array routing network.

All reconfiguration commands are issued in batches. Batches are designated with a *batchCommandBegin()* … *batchCommandEnd()* pair. Commands within the same batch are issued effectively in parallel[14]. It is the responsibility of the issuer (i.e. scheduler) to guarantee that commands within a batch do not have resource conflicts, such as loading the configuration and state for the same CP. The *batchCommandEnd()* will use simple scoreboarding to check for resource usage violations.

The purpose of this extra layer of abstraction is to ease future migration to physical hardware, as well as make the scheduler implementation less susceptible to array design changes. The actual implementation of the array is black-boxed from the scheduler.

[13] There are also several specialized API calls for modifying individual registers such as *changeSegmentMode()*, *changeSegmentTRAandPBOandMAX()*, and *resetSegmentDoneFlag()*.

[14] In reality, batched commands are issued with a one clock cycle offset because it is assumed only one command can be issued per cycle.

3 Results

Several applications were developed using the SCORE compute model. They are used to benchmark the scheduler effectiveness and demonstrate scheduler functionality. The applications that are mapped include: wavelet encoder, wavelet decoder, and JPEG encoder.

Basic performance scaling experiments were performed where the number of CPs is varied to observe the effect on application *makespan*[15]. In addition, runtime cost measurements were preformed to better understand future areas for optimization.

3.1 Application Overview

3.1.1 Wavelet Encoder/Decoder

The discrete wavelet transform used in the wavelet image encoder/decoder is a recursive operation. An image is first decomposed into low frequency and high frequency components. The decomposition is repeated on the low frequency component. The recursion goes on for as much iteration as is mandated by the compression algorithm. In our particular algorithm, this recursion is finite and known statically.

Mathematically, each decomposition passes each line of original data through a high-pass filter and a low-pass filter in parallel and down-samples the output streams of each filter by a factor of two. The total number of samples is preserved. One iteration in the above recursion consists of a horizontal decomposition followed by a vertical composition. Afterwards, the original dataset is split into four smaller datasets. The compression algorithm used performs three recursive iterations, discarding high frequency data from the first recursion. (see Figure 20).

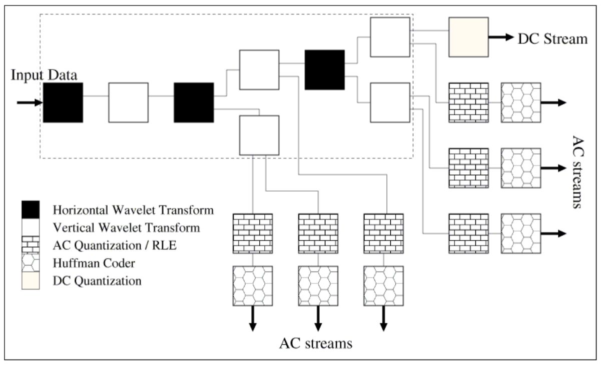


Figure 20 - Wavelet Encoder Dataflow

[15] Makespan is the number of cycles it takes to execute an application to completion.

The outputs of the discrete wavelet transform are scalar quantized. Quantization coefficients are compacted using zero-length encoding in all but the lowest frequency output where runs of zeroes are not expected. Finally, run lengths and levels are Huffman-coded into output bit-streams. In wavelet decoder, the above process is reversed, starting with the outputs from the Huffman-coders.

3.1.2 JPEG Encoder

JPEG encoder mathematically decomposes the input data into high and low frequency components. The image is first segmented into 8x8 pixel blocks. 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 8x8 block of coefficients, most of which are close to zero. The coefficients are scalar quantized and scanned into a one-dimensional stream via a *zigzag* scan. Quantized coefficients are compacted with zero-length encoding, after which runs and lengths are Huffman encoded (see Figure 21).

3.2 Performance Scaling

For these experiments, we assume a single-chip system as described in Section 1.1.2. Table 1 summarizes the parameters we assume for the system during the experiments. No limitations on routability among pages are currently modeled.

The processor code (i.e. user process, IPC thread, and scheduler thread) is executing natively on a Pentium III microprocessor running at 500 MHz with 512 Mbyte of memory. The reconfigurable array is simulated by the simulator thread on a second Pentium III microprocessor running in parallel at 500 MHz. The simulator accounts for all time required to reconfigure pages, store state, and transfer data between memories in the chip. Scheduling overhead is assumed to be 50,000 cycles and is overlapped with array execution. This allows the performance of the scheduling heuristic to be measured independent of the scheduler implementation. Section 3.3 contains measurements of the actual scheduler cost gathered from various runs.

All three mapped applications are run on a series of architecture-compatible SCORE systems with a varying number of CPs among the systems. The ratio between the number of CPs and CMBs is kept at 1:1 when possible. At the data points where this is not possible (because the number of I/Os from a single page is too great) the number of CMBs is held at the minimum possible; additionally, a dashed line is used in the graph.

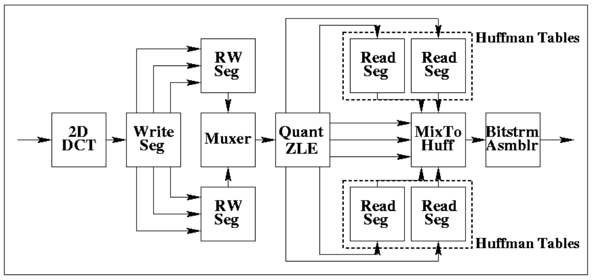


Figure 21 - JPEG Encoder Dataflow

Figure 22 shows the effect on wavelet encoder makespan as array size is varied from 1 to 30 CPs. The application is fully spatial at 30 CPs.

In general, makespan decreases monotonically with increases in the array size. The few instances violating monotonicity are believed to be artifacts of local scheduling decisions.

The performance curve is shallower between 15 CPs and 30 CPs than between 1 CP and 15 CPs. This is caused by CP under-utilization from the horizontal and vertical transform stages (see Figure 20). Each stage accepts an input token stream at rate of N and outputs two output token streams, each at a rate of N/2. This reduction in token stream rate propagates to subsequent stages so that in the longest path, the rate is reduced to N/32. Between 15 CPs and 30 CPs, the scheduling overhead and reconfiguration cost is offset by increased CP utilization (through time multiplexing). As the number of CPs is reduced further, the physical CPs become fully utilized and scheduling overhead begins to adversely affect the total makespan. In addition, because the number of CMBs also shrinks, the design becomes CMB bound. This accounts for the steeper performance curve as array size approaches 1 CP.

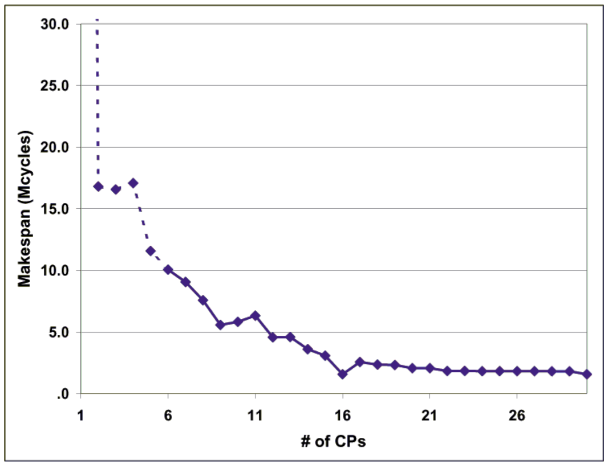


Figure 22 - Wavelet Encoder: Makespan vs. Array Size (64-LUT CPs)

Simulator Parameters	**Value Assumed**
Reconfiguration Time	5,000 cycles
Scheduler Timeslice	250,000 cycles
CP Size	64/512-LUTs
CMB Size	2 Mbits
External Memory Bandwidth	2 GBytes/s

Table 1 - System Parameters for Experiments

Three points can be derived from this graph:

?? The scheduler is able to automatically schedule a SCORE application onto less hardware while maintaining a reasonable area-time curve (CP-makespan);

?? Not all CPs may be fully utilized in a fully spatial implementation of a design;

?? The scheduler is able to automatically exploit CP under-utlization and effectively time-multiplex the design on to reduced hardware.

Figure 23 shows the effect on wavelet decoder makespan as array size is varied from 1 to 27 CPs. The application is fully spatial at 27 CPs. Since wavelet decoder is similar to wavelet encoder, its performance graph exhibits some of the same features as wavelet encoder.

As with wavelet encoder, the performance graph for wavelet decoder is shallower between 15 CPs and 27 CPs due to CP under-utilization. The non-monotonicity between 5 CPs and 15 CPs is more pronounced in wavelet decoding, but follows the same general increases due to scheduling overhead and CMB bounds.

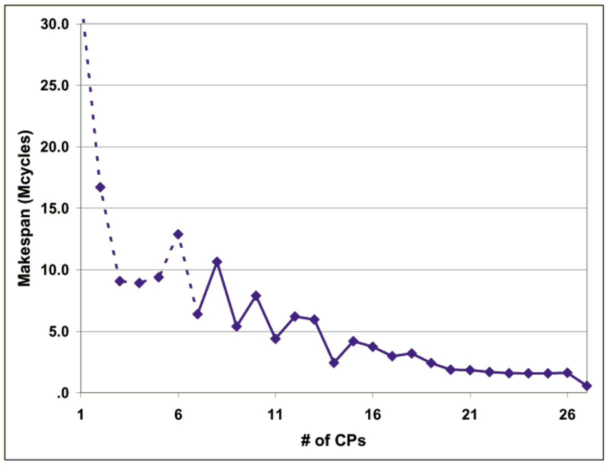


Figure 23 - Wavelet Decoder: Makespan vs. Array Size (64-LUT CPs)

Figure 24 shows the effect on JPEG encoder makespan as array size is varied from 1 to 13 CPs. The JPEG encoder implementation is different from the wavelet encoder/decoder because it is implemented using 512-LUT CPs versus 64-LUT CPs. Another difference with JPEG encoder is that its pages require a large number of I/O streams. The largest number of streams a page requires is 16. This is larger than the number of pages in the design, making it impossible to maintain a 1:1 CP-to-CMB ratio (see Section 4.4 for future work that may alleviate this problem). Therefore, all JPEG encoder performance numbers reflect a constant array CMB count of 16.

As with wavelet encoder/decoder, the performance graph for JPEG encoder is largely decreases monotonically as the array size increases. However, it does not experience the shallower performance curve when nearly fully spatial. In fact, as seen in Figure 24, JPEG encoder experiences significant performance degradation immediately upon becoming non-fully spatial. The large number of I/O streams incident to certain pages causes a large number of CMBs to be consumed. Therefore, JPEG encoder becomes CMB bound much earlier than either wavelet encoder or decoder.

3.3 Scheduler Runtime Cost Analysis

Empirical measurements were done of the scheduler runtime cost to better understand the overhead of automatic scheduling for reconfigurable devices. In these measurements, per timeslice scheduling cost is broken down into individual scheduling stages (see Section 2.3.2).

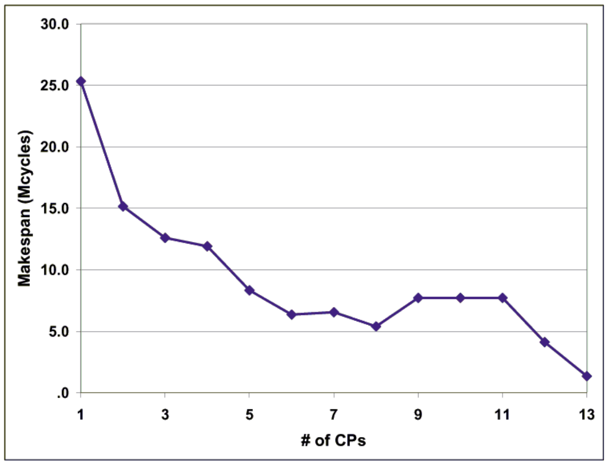


Figure 24 - JPEG Encoder: Makespan vs. Array Size (512-LUT CPs)

getArrayStatus(), *issueReconfigCommands()*, and *performCleanup()* are omitted in the measurements. *getArrayStatus()* and *issueReconfigCommands()* are omitted because they consist largely of hardware API calls. These calls do not contribute meaningfully to the scheduler decision. *performCleanup()* is omitted since it is not part of the scheduler timeslice critical path.

Measurements are performed using each of the three mapped application to discover the effect of design dataflow on the runtime cost. The measurements are run on the same series of architecture-compatible SCORE systems with a varying number of CPs among the systems.

The SCORE runtime executable is compiled using –O3 compiler optimizations. The measurements are obtained using the "x86 Performance-Monitoring Counters for Linux"[16] library. This library utilizes performance counters in Intel Pentium microprocessors and measures the number of clock cycles taken for a particular section of code. Overhead caused by instruction and data cache misses are included in this measurement.

Figure 25 shows the breakdown of scheduler runtime cost as a function of array size when running the wavelet encoder application. The scheduler cost per timeslice ranges from about 60,000 cycles to 260,000 cycles. The three major contributors appear to be *gatherStatusInfo()*, *schedulerClusters()*, and *performPlacement()*.

In general, *scheduleClusters()* consumes at least half of the scheduler overhead. The only time this is not true is when the design is fully spatial. The current scheduler implementation does not special case a fully spatial implementation. Therefore, time is still spent discovering that no rescheduling needs to be performed.

An examination of Figure 25 suggests that a less complex scheduling heuristic needs to be investigated. This would reduce the cost contributed by *scheduleClusters()*. One possible candidate is static scheduling (see Section 4.3), which may reduce overhead in several stages, including *gatherStatusInfo()*, *findFreeableClusters()*, and *scheduleClusters()*.

An important issue to note is that the scheduling overhead increases almost monotonically with increases in array size. Since the scheduler is required to fill more physical CPs and CMBs, this increase is expected. However, this increase is sub-linear with the number of

[16] http://www.csd.uu.se/~mikpe/linux/

physical CPs. This may be evidence that the timeslice-scheduling model is valid for reconfigurable devices. Making batch scheduling decisions at timeslice boundaries has many advantages, including amortizing the overhead of status gathering and context switching. In addition, batch scheduling offers the potential opportunity to reduce stitch buffering.

Figure 26 shows a similar breakdown of scheduler runtime cost when running the wavelet decoder application. Since the two applications are similar in overall design, their effects on runtime cost are similar. The main difference that exists is the absence of the *dealWithDeadLock()* stage through much of the wavelet decoder run. Unlike wavelet encoder, wavelet decoder does not contain streams susceptible to physical stream buffer limitations. The occurrence of deadlock detection/resolution between 9 CPs and 19 CPs is most likely the result of stale array status causing premature deadlock detection.

Finally, Figure 27 shows the breakdown of scheduler runtime cost when running the JPEG encoder application. Unlike wavelet encoder and decoder, JPEG encoder has relatively few pages to schedule. Therefore, one would expect *scheduleClusters()* to consume less time. However, from the figure, overall scheduling time has increased.

The reason for this increase is a result of the “frontier” scheduling heuristic (see Section 2.4.1) The “frontier” scheduling heuristic is similar to a breadth-first-search and is linear on the number of nodes and edges in the graph. JPEG encoder contains pages with as many as 16 I/O streams incident. Wavelet encoder and decoder pages have a maximum of 6 to 7 I/O streams incident. This increase in I/O streams in the cause for increased *scheduleClusters()* cost.

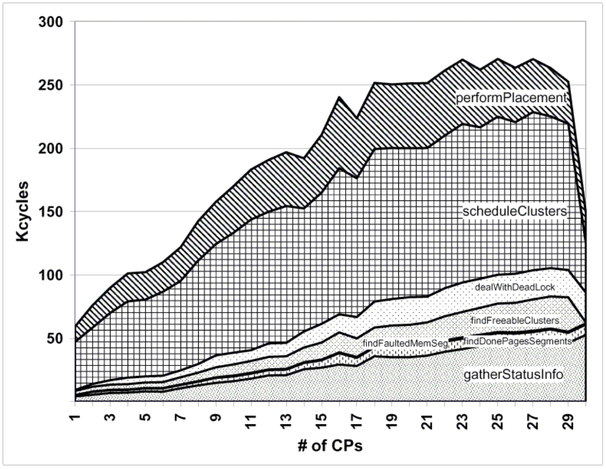


Figure 25 - Breakdown of Per Timeslice Scheduler Cost (Wavelet Encoder Execution)

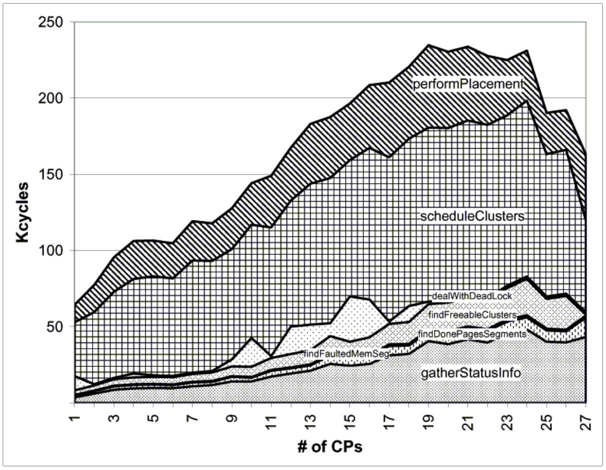


Figure 26 - Breakdown of Per Timeslice Scheduler Cost (Wavelet Decoder Execution)

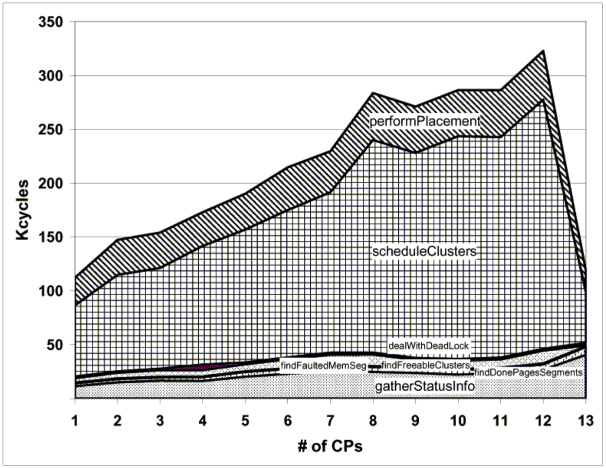


Figure 27 - Breakdown of Per Timeslice Scheduler Cost (JPEG Encoder Execution)

4 Future Work

In the process of designing and implementing the SCORE scheduler, several areas for future work have been identified. The areas of future work range from optimizing scheduling time and result quality to fairness guarantees for multiple simultaneous designs.

4.1 Expandable Stitch Buffers

Stitch buffers are added into the design dataflow to retime data tokens between non-coresident pages and segments. They are also used to resolve bufferlock introduced by finite physical stream buffering. The current implementation allows for at most one stitch buffer between pages and segments. Additionally, the size of individual stitch buffers is fixed and must fit completely within a single CMB.

Ideally, we would like the size of a stitch buffer to be expandable at runtime. One method of providing this is by chaining together fixed-size stitch buffer. Figure 28 shows an example of an expandable stitch buffer implemented as a chain of fixed size stitch buffers.

The expandable stitch buffer has been expanded to a size of 10 Mbits while the physical CMB size is 2 Mbits. The expandable stitch buffer starts out as a single 2 Mbit fixed-size stitch buffer (0-2 Mbit) simultaneously read from by page 1 and written to by page 0. As the expandable stitch buffer becomes full, the scheduler creates another 2 Mbit fixed size stitch buffer (2-4 Mbit). Page 1 continues to read from (0-2 Mbit) while page 0's output stream is rerouted to write to (2-4 Mbit). (0-2 Mbit) and (2-4 Mbit) are not connected directly together and function strictly as sequential read-only and sequential write-only segments, respectively.

As the expandable stitch buffer becomes filled again, additional 2 Mbit fixed-size stitch buffers are created and added to the chain as shown in Figure 28. Only the head and the tail of the stitch buffer chain are read from or written to. The intermediate fixed-size stitch buffers are not resident on the array. The chain is collapsed as the head stitch buffer (i.e. (0-2 Mbit)) is completely drained. The next stitch buffer then become the head of the chain (i.e. (2-4 Mbit)).

4.2 Quantized Priority "Frontier" Cluster List

From section 2.4.1, we see that the "frontier" cluster list is managed as a priority list. The current "frontier" cluster list is implemented as a binary heap [CORMEN96]. The result is an insertion time of *O(log(N))*, *N* being the number of clusters scheduled in the system.

However, the level of granularity offered by a full heap implementation may not be necessary. An alternate implementation is a quantized priority list. Arbitrary quantization levels are defined (i.e. low, medium, high) and assigned a priority subrange (i.e. low = 0 to 85000, medium = 85001 to 175000, high = 175001 to 250000). The quantized priority list is implemented as an array of cluster linked lists. The size of the array is defined as the number of priority levels. Within each level, the clusters in the linked lists are unordered. Figure 29 is an example of the quantized priority list described. Clusters A, B, C, D, E, and F are shown to be organized in the three-level priority list with high, medium, and low priorities.

The advantage of a quantized priority "frontier" cluster list an insertion time of *O(1).* Once the raw priority of a cluster is determined, a simple comparison yields the quantization level for the cluster. The new cluster is simply appended to the end of the appropriate list. When the scheduler wishes to remove a cluster with the highest priority, it simply finds a priority level with a non-empty cluster list and removes the head of the list.

4.3 Static Scheduling in SCORE

In the current scheduler implementation, scheduling decisions are dynamically made at runtime based on runtime statistics gathered from the array. Dynamic scheduling decisions allow the SCORE scheduler to cope with dataflow with data-dependent data rates or dynamically constructed dataflows. However, there are situations where some or all of the dataflow data rate is static. It would be desirable to pre-compute the scheduling order in this situation. In particular, following a pre-computed scheduling "recipe" would reduce scheduling runtime.

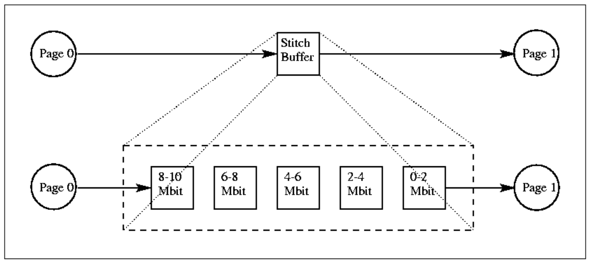


Figure 28 - Example of Expandable Stitch Buffer Implementation

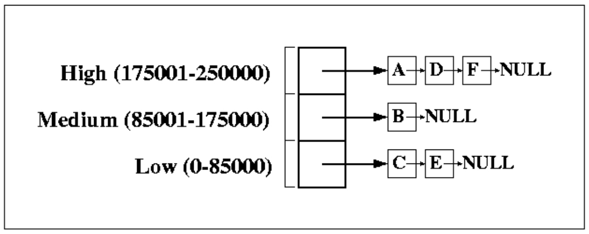


Figure 29 - Example of Quantized Priority "Frontier" Cluster List Implementation

However, two questions remain:

?? When should the static schedule be calculated?

?? What should the static schedule specify?

The concern is that, in a general SCORE environment, two potentially unknown variables can affect static scheduling: (i) varying array size, and (ii) multiple SCORE designs executing simultaneously. If the static schedule is calculated at compile time and specifies exactly when each page and segment should be scheduled along with any necessary stitch buffers, the schedule cannot cope with array size variations or interference from other user designs.

If, instead, the scheduler calculates the static schedule at operator instantiation time, the schedule can cope with array size. However, an overhead is incurred at each operator instantiation. Both solutions cannot cope with interference from other design running simultaneously on the array. In addition, the above solutions cannot take advantage of dataflow graphs where a subset of the dataflow is static data rate. Figure 30 shows an example of a dataflow where subsets of the dataflow, such as (A, B, C) and (F, G), are internally static data rate, while the entire dataflow is dynamic data rate.

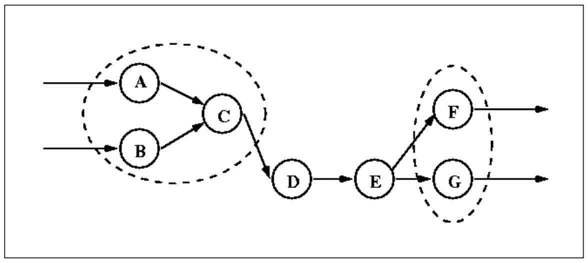


Figure 30 - Example of Subsets of Nodes Marked as Static Rate Dataflow

Therefore, a general static scheduling solution cannot depend on the knowing the available compute resources on the array. While an optimal solution cannot be guaranteed, this means that no restrictions are placed on the SCORE compute model. A possible compromise solution is to have the static schedule only specify the relative ordering of the pages and segments within each scheduling subset. For example, in Figure 30, a static schedule for (A, B, C) might specify that the order of scheduling is (i) A, (ii) B, (iii) C. Each static scheduling subset is then collapsed into a single black-boxed node when performing dynamic scheduling. Whenever the dynamic scheduler determines it will schedule one of the black-boxed static subsets, the pre-computed schedule is used.

Some issues that need to be considered include:

?? How will the dynamic scheduler view the black-boxed static subsets? (i.e. how many array resources will it be viewed as consuming?)

?? Once a black-boxed static subset is encountered, how will the precomputed schedule be interleaved with the general dynamic schedule?

These issues are not addressed in this report. Their resolution will be a part of any future static-dynamic SCORE scheduler.

4.4 Min-edge-cut Clustering

Currently, the SCORE scheduler uses the cluster abstraction to deal with the pathological case of feedback loops in the dataflow. This is handled in *addOperator()* where nodes of a feedback loop are placed within a cluster to guarantee coscheduling (see Section 2.3.1.1).

Another potential use of clustering is to reduce runtime CMB usage. For example, in Figure 31 the top dataflow shows a subset of a larger dataflow. This subset is purely feed-forward. Using the current clustering mechanism, each node is placed within its own cluster. However, this results in a large number of streams between clusters 1 and 2. If cluster 1 and cluster 2 in the top dataflow become non-coresident, the stitch buffering between cluster 1 and 2 would consume three CMBs on the array. If the array contains few CMBs, the design would quickly become CMB bound. This suggests that cluster 1 and 2 should always be coscheduled. The easiest way to achieve this is to guarantee that node B and C reside in the same cluster (see Figure 31 bottom dataflow).

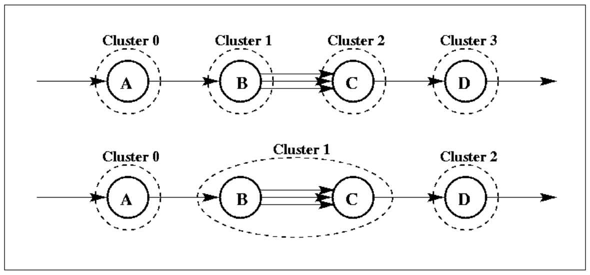


Figure 31 - Example of Possible Min-edge-cut Clustering

To implement this method of clustering, an algorithm based on network flow techniques can be employed. Max-flow min-cut algorithms exist that find a min-cut bipartition in polynomial time ([EVEN79] [FORD62] [HU85] [LAWLER76]). In [YANG94], Yang and Wong propose a balanced bipartition heuristic based on repeated max-flow min-cut techniques. By applying these max-flow methods, a SCORE operator could be partitioned into clusters with a minimum number of I/O streams.

4.5 Process Fairness Guarantees

The current scheduler implementation includes a one-level priority scheme. Priorities are calculated for each cluster and the clusters are sorted using a binary heap. This heuristic works well when there is only one SCORE application running.

However, when multiple SCORE applications are executing simultaneously, a one-level priority scheme does not guarantee fairness among the processes. While each process will eventually execute to completion (i.e. no process will starve), a process with a large number of pages or segments can monopolize array resources. The current scheme makes no attempt to prioritize clusters based on the parent process.

There are various methods for approaching this problem. As the number of application increases, the desired behavior is for the performance of each application to degrade by an equal amount. One method for achieving this is to time multiplex the array among the processes. Given *N* applications, an allocation cycle of *N* slots is defined. Each application is assigned one slot. During its assigned slot, an application has complete access to the array. Each slot is one or more timeslices in duration. The “frontier” scheduling heuristic may still be used. Each process would contain its own “frontier” cluster list that is consulted during its assigned time slot. The disadvantage of this method is that a process with little or no work to perform still occupies an equal slot as active processes.

An alternative to strict time multiplexing of the array is a modified cluster priority scheme. The current cluster priority can be augmented with process priority. The process priority scheme can be one of various scheduling techniques, such as lottery scheduling [WALDSPURGER94]. The exact method for providing process fairness in SCORE remains to be determined in future implementations.

5 Conclusion

We have presented an implementation of a dynamic runtime array scheduler for the SCORE compute model. Work presented indicates that dynamic scheduling of designs onto varying-size reconfigurable hardware is possible. In addition, the scheduler is able to time-multiplex designs onto reduced hardware and still achieve acceptable area-time curves by automatically exploiting resource under-utilization (i.e. wavelet encoder and decoder). Through scheduler cost analysis, we have found that the current implementation does not yet meet the 50,000 cycle overhead goal. However, several ideas have been discussed that may help reduce the overhead, including: static scheduling for subset dataflows and priority list quantization. Overall, this project is successful in its goal to show that dynamic and automatic scheduling of designs onto reconfigurable hardware is indeed possible using the SCORE compute model. The current implementation serves as an important framework for future scheduler development in SCORE.

6 Appendix – Scheduler Data

The SCORE scheduler uses several specialized data types and structures to organize user designs and maintain the scheduler's task lists. In this section, we describe the most important and common data types as well as the key data structures used by the scheduler.

6.1 Data Types

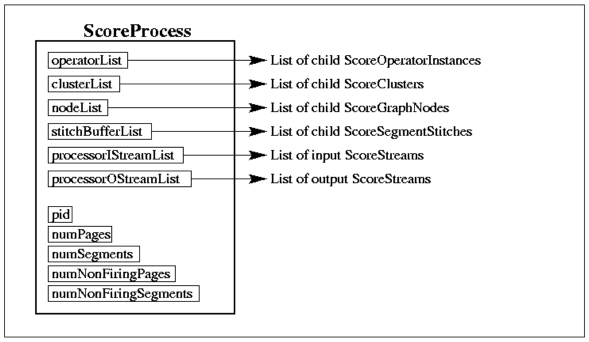
The user's design is maintained in a hierarchy of C++ objects. The hierarchy levels include: ScoreProcess, ScoreOperatorInstance, ScoreCluster, ScoreGraphNode and ScoreStream. At the highest level, a ScoreProcess is directly mapped to a user SCORE application. At the lowest level, a ScoreGraphNode and ScoreStream are the dataflow graph nodes and edges. The details of each data type are described in this section.

ScoreProcess resides at the highest level of the design hierarchy. An instantiation of ScoreProcess is created every time a new SCORE application is linked to the runtime. The scheduler maintains a list of ScoreProcess objects indexed process ID. All instantiated operators, clusters, pages, and segments for a particular process are reachable by traversing ScoreProcess data structures. During the normal course of scheduling, the ScoreProcess list is not accessed. However, this list is important during deadlock detection/resolution (see Section 2.4.3), which is performed on an entire process at a time. In the future, the process list may be used to provide fairness and performance guarantees among multiple SCORE applications.

6.1.1 ScoreProcess

Figure 32 shows the important elements of the ScoreProcess data type. As the top level of the hierarchy, it contains lists of all components of an application, including instantiated operators, clusters, nodes, and streams. It also contains lists of streams that are written to or read from by the processor. Each of the instantiated operators, clusters, and nodes have a parent ScoreProcess pointer pointing back to this data structure.

In addition to the component and stream lists, ScoreProcess also stores the process ID of the SCORE application. In the event the user decides to instantiate the design using separate operators, this allows the scheduler to match operators with an existing SCORE process. Finally, ScoreProcess contains a count of the number of pages and segments as well as the number of non-firing pages and segments since the last timeslice. These variables are used to determine when to run deadlock detection (see Section 2.4.3).



ScoreProcess
operatorList
clusterList
nodeList
stitchBufferList
processorIStreamList
processorOStreamList
pid
numPages
numSegments
numNonFiringPages
numNonFiringSegments
List of child ScoreOperatorInstances
List of child ScoreClusters
List of child ScoreGraphNodes
List of child ScoreSegmentStitches
List of input ScoreStreams
List of output ScoreStreams

Figure 32 - Diagram of ScoreProcess Data Type

6.1.2 ScoreOperatorInstance

The ScoreOperatorInstance is the instantiated data type of a user operator. It contains sufficient information for the IPC thread to accurately reconstruct the dataflow of the operator. ScoreOperatorInstance objects are only accessed during operator instantiation and operator cleanup in the *performCleanup()* stage (see Section 2.3.2.10).

Figure 33 shows that the ScoreOperatorInstance data type contains lists of the pages and segments making up the operator. The "pages" and "segments" variables indicate the number of pages and segments in the individual lists. Finally, a pointer exists to reference back to the parent ScoreProcess once the parent process has been identified.

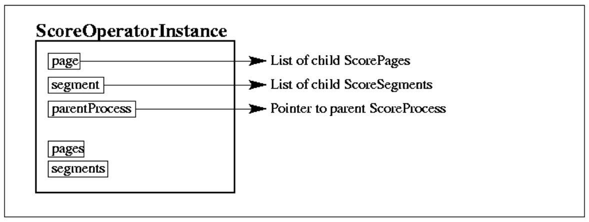


Figure 33 - Diagram of ScoreOperatorInstance Data Type

6.1.3 ScoreGraphNode

The ScoreGraphNode data type is the base class for ScorePage (see Section 6.1.4) and ScoreSegment (see Section 6.1.5). It contains the necessary variables and lists to link the node into the dataflow as well as the design hierarchy. In addition, it also provides a place to store status information concerning the node.

Figure 34 shows the important variables in the ScoreGraphNode data type. It contains a list of the node's input and output streams as well as pointers to the parent ScoreProcess, parent ScoreOperatorInstance, and parent ScoreCluster (see Section 6.1.8). The exact number of input and output streams is indicated in the "inputs" and "outputs" variables. Variables also exist to store node status, such as: if the node has signaled done, if the node is resident, and if the node is being scheduled on the array. If the node is resident, its location on the array is stored in residentLoc. Finally, didNotFireLastResident is set if the node did not consume or produce tokens during the last timeslice it was resident on the array. The scheduler uses this to determine when to perform deadlock detection (see Section 2.4.3).

Objects of type ScoreGraphNode are never instantiated directly by the runtime system. Rather, the system instantiates the derivative types: ScorePage and ScoreSegment.

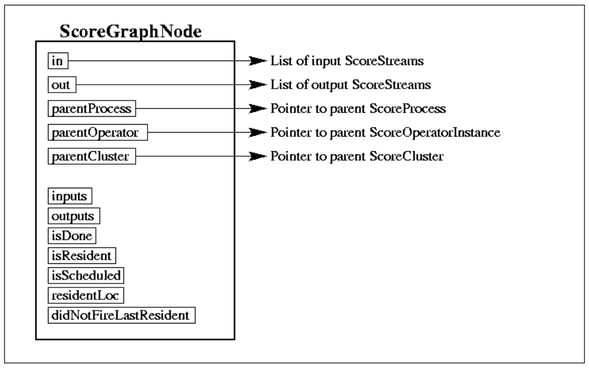

ScoreGraphNode
in
out
parentProcess
parentOperator
parentCluster
inputs
outputs
isDone
isResident
isScheduled
residentLoc
didNotFireLastResident
List of input ScoreStreams
List of output ScoreStreams
Pointer to parent ScoreProcess
Pointer to parent ScoreOperatorInstance
Pointer to parent ScoreCluster

Figure 34 - Diagram of ScoreGraphNode Data Type

6.1.4 ScorePage

The ScorePage data type is derived from ScoreGraphNode. It is a specialized version of ScoreGraphNode used to represent SCORE pages in the dataflow. Objects of this type are first created during operator instantiation (see Section 2.3.1).

6.1.5 ScoreSegment

The ScoreSegment data type is also derived from ScoreGraphNode. It is a specialized version of ScoreGraphNode used to represent SCORE segments in the dataflow. It also serves as a parent class for ScoreSegmentStitch (see 6.1.7). Objects of this type are first created either during operator instantiation (see Section 2.3.1) or as a result of stitch buffer insertion (see 2.3.2.7).

In addition to the variables in ScoreGraphNode, ScoreSegment contains information pertinent to segments: a pointer to the user data block for the segment, the size of the data block, and the mode of the segment. The data block pointer and size allow the scheduler to properly load the CMBs on the array when the segment is scheduled. The mode can be any one of the modes described in Section 1.1.1.2). For most segments, the mode is constant throughout the instantiation of the segment. However, with ScoreSegmentStitch, the segment mode may change depending on the resident dataflow nodes.

6.1.6 ScoreStream

While ScoreGraphNode describes the nodes of the dataflow graph, ScoreStream describes the token stream connections between pages and segments. Figure 35 shows the key variables in ScoreStream. The most important variables are src, srcNum, sink, and snkNum which reference the producer and consumer nodes along with the corresponding output and input port numbers. In addition to the source and sink node pointers, ScoreStream also stores information about the type (page, segment, or processor) of the nodes in srcFunc and snkFunc. This optimizes graph traversal by providing type information without dereferencing src or sink. Further optimization is provided by the srcIsDone and sinkIsDone flags which indicate whether the source and sink nodes have signaled done. Finally, isCrossCluster and inProcessorArrayStream indicate whether the stream connects nodes in different clusters or if it provides processor-array communication.

6.1.7 ScoreSegmentStitch

The ScoreSegmentStitch data type is derived from the parent class ScoreSegment. It serves to represent a stitch buffer added by the scheduler to the dataflow (see Section 2.3.2.7). Currently, its key variables are the same as ScoreSegment. When expandable stitch buffers are implemented, variables relevant to the expanded stitch buffer will also be stored in this data type (see Section 4.1).

The role of a stitch buffer is to capture the output data from a stream whose nodes are not coscheduled. It saves the output tokens until the stream consumer can run in the array. The stitch buffer then provides the previously captured tokens to the consumer. This helps maintain the abstraction of infinite hardware resources.

6.1.8 ScoreCluster

The ScoreCluster data type is used to represent a cluster of pages or segments. In SCORE, nodes in a cluster are scheduled atomically; this means either all of these nodes are resident or all are not resident. Clusters are formed as part of the operator instantiation process. To guarantee nodes of a feedback loop are never non-coresident (see Section 2.3.1.1). In the future, clusters may also be used to optimize scheduling in other ways, such as minimizing CMB usage (see Section 4.4).

ScoreCluster includes a list of member ScoreGraphNodes, along with lists of ScoreStreams that form the cluster input and output ports. There is also a pointer to the parent ScoreProcess. Complementing nodeList, there are counts of the number of individual pages and segments in the cluster. There are also flags for whether the cluster is resident, scheduled, or considered freeable. Finally, the isFrontier, isHead and lastFrontierTraversal variables are specialized fields used by the "frontier" scheduling heuristic (see Section 2.4.1).

6.2 Data Structures

The SCORE scheduler maintains several data structures that allow it to keep track of the state of the array as well as facilitate transferring decisions between scheduler stages (see Section 2.3.2).

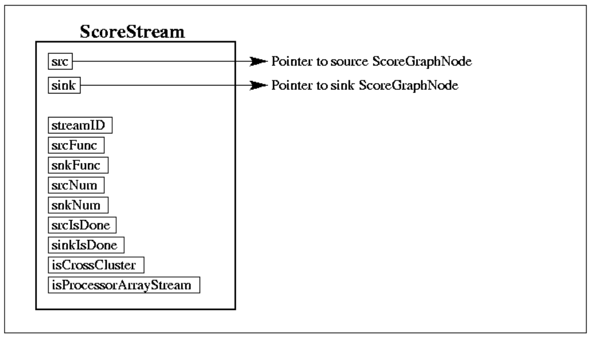


Figure 35 - Diagram of ScoreStream Data Type

Figure 37 shows a diagram of the important data structures in ScoreScheduler. The entire design hierarchy can be accessed via the list of ScoreProcesses, processList. The scheduling task list and resident cluster list consist of frontierClusterList, headClusterList, waitingClusterList, and residentClusterList. These are used by the "frontier" scheduling heuristic (see Section 2.4.1). The streams used to communicate between the processor and array are stored in processorIStreamList and processorOStreamList. Any stitch buffers added to handle non-coresident streams as well as those added to resolve bufferlock are stored in stitchBufferList.

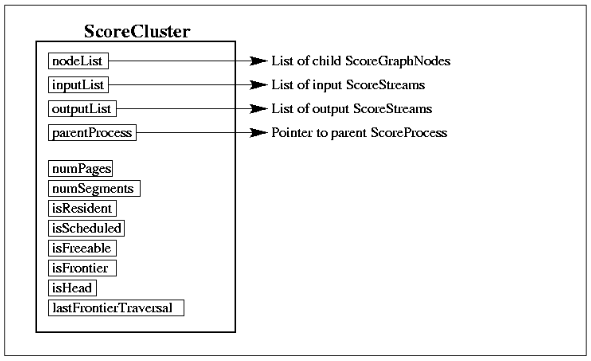


Figure 36 - Diagram of ScoreCluster Data Type

As the scheduler schedules and removes pages and segments, arrayCP and arrayCMB are updated to reflect the resident nodes on the array. These are arrays whose elements consist of ScoreArrayCP and ScoreArrayCMB. The main fields in ScoreArrayCP and ScoreArrayCMB are:

?? Active (the page or segment that is resident at this location).

?? Scheduled (the page or segment that is scheduled to be resident at this location).

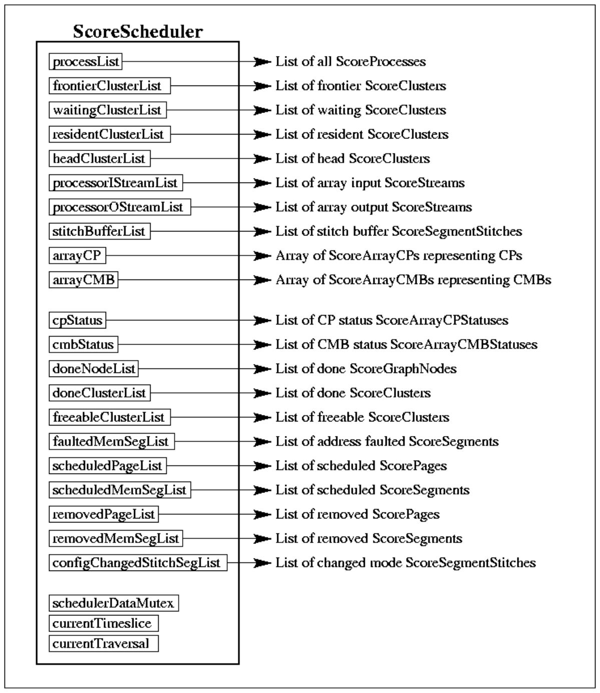

ScoreScheduler
processList
frontierClusterList
waitingClusterList
residentClusterList
headClusterList
processorIStreamList
processorOStreamList
stitchBufferList
arrayCP
arrayCMB
List of all ScoreProcesses
List of frontier ScoreClusters
List of waiting ScoreClusters
List of resident ScoreClusters
List of head ScoreClusters
List of array input ScoreStreams
List of array output ScoreStreams
List of stitch buffer ScoreSegmentStitches
Array of ScoreArrayCPs representing CPs
Array of ScoreArrayCMBs representing CMBs
cpStatus
cmbStatus
doneNodeList
doneClusterList
freeableClusterList
faultedMemSegList
scheduledPageList
scheduledMemSegList
removedPageList
removedMemSegList
configChangedStitchSegList
List of CP status ScoreArrayCPStatuses
List of CMB status ScoreArrayCMBStatuses
List of done ScoreGraphNodes
List of done ScoreClusters
List of freeable ScoreClusters
List of address faulted ScoreSegments
List of scheduled ScorePages
List of scheduled ScoreSegments
List of removed ScorePages
List of removed ScoreSegments
List of changed mode ScoreSegmentStitches
schedulerDataMutex
currentTimeslice
currentTraversal

Figure 37 - Diagram of data structures in ScoreScheduler

These arrays are used when making scheduling decisions as well as during array reconfiguration.

The scheduler also maintains data structures that are used to communicate decisions between scheduler stages (see Section 2.3.2). These include:

?? cpStatus and cmbStatus: current array status from *getArrayStatus()*;

?? doneNodeList and doneClusterList: nodes and clusters considered done by *findDoneNodePagesSegments()*;

?? freeableClusterList: clusters considered freeable by *findFreeableClusters()*;

?? faultedMemSegList: segments determined to have experienced an address fault by *findFaultedMemSeg()*;

?? scheduledPageList, scheduledMemSegList, removedPageList, removedMemSegList, configChangedStitchSegList: pages and segments scheduled or removed by *scheduleClusters()*; in addition, any resident stitch buffers needing mode adjustment.

The final variables in the scheduler are schedulerDataMutex, currentTimeslice, and currentTraversal. The schedulerDataMutex is a lock variable used to synchronize between the IPC thread and the scheduler thread for access to the scheduler data structures. The currentTimeslice variable allows the scheduler to keep track of the number of timeslices that have elapsed. Finally, the currentTraversal variable is used by the “frontier” scheduling heuristic to prevent scheduling starvation (see Section 2.4.1).

7 Bibliography

[BHATTACHARYYA95] S.S. Bhattacharyya, S. Sriram, E.A. Lee. Minimizing Synchronization Overhead in Statistically Scheduled Multiprocessor Systems. In *Proceedings of the International Conference on Application Specific Array Processors*, 1995.

[BHATTACHARYYA96] S.S. Bhattacharyya, P.K. Murthy, E.A. Lee. *Software Synthesis From Dataflow Graphs*, Kluwer Academic Publishers, Norwell, Massachussets, 1996.

[BUCKLEE92] Joseph T. Buck, Edward A. Lee. The Token Flow Model. Presented at *Data Flow Workshop*, Hamilton Island, Australia, May 1992.

[BUCKLEE93] Joseph T. Buck, Edward A. Lee. Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model. In *Proceedings of ICASSP'93*, Minneapolis, Minnesota, April 1993.

[BUCK93] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs With Bounded Memory Using the Token Flow Model*, Technical Memorandum UCB/ERL M93/69 (Ph.D. thesis), EECS Dept., University of California, Berkeley, 1993.

[BUCK94] Joseph T. Buck, Static Scheduling and Code Generation from Dynamic Dataflow Graphs with Integer-valued Control Streams. In *Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, Oct. 30-Nov. 2, 1994.

[CASPI00] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, André DeHon, John Wawrzynek. Stream Computations Organized for Reconfigurable Execution (SCORE): Introduction and Tutorial. To appear in *Proceedings of the 10th International Workshop on Field Programmable Logic and Applications (FPL'00)*, August 2000.

[CORMEN96] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts, 1996.

[DEHON96] André DeHon. *Reconfigurable Architectures for General-Purpose Computing*, AI Technical Report 1586 (Ph.D. thesis), MIT Artificial Intelligence Laboratory, 545 Technology Sq., Cambridge, MA 02139, October 1996. http://www.ai.mit.edu/people/andre/phd.html

[EVEN79]	S. Even. *Graph Algorithms*. Computer Science Press, 1979.
[FORD62]	J. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.
[FRANKE96]	H. Franke, P. Pattnaik, L. Rudolph. Gang Scheduling for Highly Efficient Distributed Multiprocessor Systems. In *Proceedings of the 6th Symposium on Frontiers of Massively Parallel Computation (Frontiers '96)*, Annapolis, Maryland, Oct. 27-31, 1996.
[GAJSKI92]	D.D. Gajski, N.D. Dutt, A.C-H. Wu, S.Y-L. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Boston, Massachusetts, 1992.
[HA97]	Soonhoy Ha, Edward A. Lee. Compile-Time Scheduling of Dynamic Constructs in Dataflow Program Graphs. In *IEEE Trans. On Computers*, vol. 46, no. 7, July 1997.
[HAUSER97]	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.
[HENNESSY90]	John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, Inc., 1990, pp 290-299.
[HU85]	T. C. Hu and K. Moerder. *Multiterminal Flows in a Hypergraph*. Hu and Kuh eds. IEEE Press, 1985.
[KNOWLTON65]	K. C. Knowlton. "A Fast Storage Allocator". In Communications of the ACM, vol. 8, pp. 623-625, Oct. 1965.
[KNUTH73]	D. E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 2nd Ed., Reading, MA: Addison-Wesley, 1973.
[KONSTANTINIDES90]	K. Konstantinides, R.T. Kaneshiro, J.R. Tani. Task Allocation and Scheduling Models for Multiprocessor Digital Signal Processing. In *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. 38, no 12, December 1990.
[LAWLER76]	E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart & Winston, New York, 1976.

[LEE91]	Edward A. Lee. Static Scheduling of Dataflow Programs for DSP. Chapter 19 in *Advanced Topics in Data-Flow Computing*, ed. J-L. Gaudiot and L. Bic, Prentice Hall, Englewood Cliffs, New Jersey, 1991.
[LIAO94]	G. Liao, E.R. Altman, V.K. Agarwal, G.R. Gao. In *Proceedings of the 27th Annual Hawaii International Conference on System Sciences*, 1994.
[PERISSAKIS99]	Stylianos Perissakis, Yangsung Joo, Junhong Ahn, André DeHon, and John Wawrzynek. Embedded DRAM for a Reconfigurable Array. In *Proceedings of the 1999 Symposium on VLSI Circuits*, June 1999.
[PINO95]	J.L. Pino, S.S. Bhattacharyya, E.A. Lee. A Hierarchical Multiprocessor Scheduling System for DSP Applications, In *Asilomar Conference on Signals, Systems, and Computers*, October 1995.
[TAU95]	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.
[TSU99]	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.
[WALDSPURGER94]	Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First Symposium on Operating System Design and Implementation*, November 1994.
[WILLIAMSON96]	Michael C. Williamson, Edward A. Lee. Synthesis of Parallel Hardware Implementations from Synchronous Dataflow Graph Specifications. In *Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, 1996.
[YANG94]	Honghua Yang and D. F. Wong. Efficient Network Flow Based Min-Cut Balanced Partitioning. In *Proceedings of IEEE/ACM International Conference on CAD-94*, San Jose, CA, November 6-10, 1994.
[YEN95]	C. Yen, S.S. Tseng, C-T. Yang. Scheduling of Precedence Constrained Tasks on Multiprocessor Systems. In *Proceedings of the 1st International Conference on Algorithms and Architectures for Parallel Processing (ICAPP 95)*, Brisbane, Qld., Australia, April 19-21, 1995.