Quasi-Static Scheduling for SCORE

Yury Markovskiy

December 7, 2004

Abstract

Previous work introduced a dynamic compute model aimed at eliminating existing barriers to the widespread efficient exploitation of reconfigurable devices. Among other achievements this model decoupled application design-time decisions from the run-time physical resource bindings. The compute model uses graphs of compute pages and memory blocks connected by stream links to capture the definition of a computation abstracted from the detailed hardware size. An automatic run-time scheduler is a required component in this compute model in that it selects the temporal sequencing of virtual resources onto the physical device, allocates hardware resources, and configures the device. Although such a scheduler could be computationally expensive, this work describes a quasi-static scheduling strategy that dramatically reduces run-time overhead without restricting the full semantic power of the dynamic dataflow graphs. This work describes the quasi-static scheduling system, analyzes the trade-offs involved in selecting a scheduler implementation, and highlights critical algorithms. It pays particular attention to the temporal partitioning of compute graphs and the management of live computation state.

Contents

Chapter 1

Introduction

As demonstrated in previous works for a variety of applications, reconfigurable computing devices such as FPGAs can achieve 10x–100x gains in performance and functional density over microprocessors [DeH96]. Yet microprocessors continue to be more popular in use due to software compatibility and automatic performance scaling across device generations. SCORE addressed these problems with an abstraction layer analogous to the function of Instruction Set Architecture (ISA) for processors. It decoupled application design-time decisions from the run-time physical resource bindings [CCH+00].

This layer of abstraction provides an opportunity for hardware mapping decisions to be deferred to a run-time scheduler, which adapts those decisions to a specific hardware configuration. Thus without application recompilation or redesign, a scheduler can enable compatibility and automatic scaling of application performance across device generations. Unfortunately, run-time scheduling for a powerful SCORE dynamic programming model could be computationally expensive.

This work presents a novel quasi-static scheduling methodology that contains the run-time scheduling overhead without restricting the full semantic power of SCORE. The work analyzes techniques for computation temporal partitioning and management of live computation state that allow the scheduler to contain device configuration overheads as well. Together these make virtualized application execution *efficient* on both small and large devices, and provide a scalable system for generations of hardware.

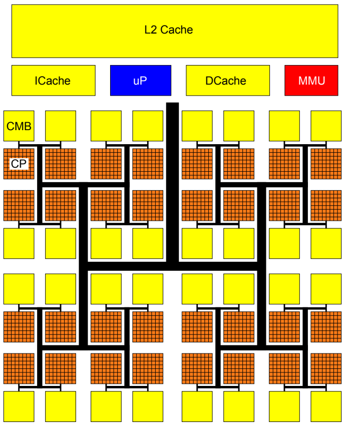


Figure 1.1: Hypothetical, single-chip SCORE system.

1.1 SCORE

[CCH$^+$00] presents SCORE (Stream Computations Organized for Reconfigurable Execution), a system that strives to eliminate existing barriers to the widespread efficient exploitation of reconfigurable devices. SCORE introduced a compute model based on paged virtual hardware, which acts in a manner similar to virtual memory. The paged model provides a framework for device size abstraction, automatic runtime reconfiguration, binary compatibility among page-compatible devices, and automatic performance scaling on larger devices without recompilation.

SCORE allows a programmer to describe a computation as a graph of *arbitrary sized* operators that communicate tokens through streams with logically unbounded buffering capacity. A high-level language compiler maps a given arbitrary sized computation into a graph of *fixed-size* compute pages constrained by the underlying architecture. In this work, the target hardware is a microprocessor and a reconfigurable array hybrid shown in Figure 1.1. The array is partitioned into *fixed-size* compute pages (CPs) and configurable memory blocks (CMBs). The run-time scheduler time-multiplexes compute pages onto physical CPs and manages buffer allocation in CMBs. This presents to the user the illusion of unbounded hardware.

SCORE presents several models that define for the developer the expected computational and execution semantics. These models expose varying levels of detail.

SCORE Programming Model. A SCORE computation is described by a dataflow graph of operators and memory blocks that communicate through streams. Streams are unidirectional, one producer, one consumer FIFO queues with logically unbounded depth. The operators consist of a finite state machine (FSM) that controls a computational datapath. A FSM consumes data tokens from one or more inputs, feeds them to the data path and emits the results. Each operator FSM has a special *done* state to indicate when a SCORE system can remove it from the computation.

The memory blocks in the graphs can operate in either sequential or random access mode. The memory access operations include read, write, or both concurrently. The operators communicate with the memory blocks *only through streams*. Depending on the mode, an operator sends an address and/or a data token to the memory block and receives the data requested. This arrangement avoids race conditions in memory accesses because a stream has only one source.

SCORE operator FSMs and memory blocks have deterministic execution semantics with blocking read and non-blocking write operations, similar to those in Kahn Process Networks [Kah74]. Operators perform computations and/or a state transitions only when the tokens on all required inputs are available. This behavior in conjunction with logically unbounded stream depth ensures that execution correctness is insensitive to stream communication latency. This makes the computation results deterministic and independent of operator ordering in any schedule.

For examples of SCORE applications, refer to Appendix A.

SCORE Execution Model. SCORE graph operators can be of arbitrary size and thus are unsuitable for direct mapping onto hardware. A high-level compiler transforms a graph of operators and memory blocks into a graph of compute pages and memory segments by partitioning and repacking the operators into pages. The underlying hardware imposes area and I/O bound requirements that the compute graph page must satisfy. That is, a virtual compute graph page must fit onto a physical page. The page firing and execution semantics are identical to those in the operators. They are deterministic and insensitive to stream communication latency.

A high-level compiler converts arbitrarily sized memory blocks into segments that fit in physical memory blocks on the hardware. Should the user-specified memory block require more contiguous memory than available on chip, the run-time system loads only a part of the memory block on chip at any moment of time. The run-time system services segmentation faults in a manner similar to a conventional operating system by bringing requested memory pages from off-chip memory.

The pages and segments communicate only through streams, which in the Execution Model have a finite, limited, hardware-specified buffering capacity. The run-time scheduler uses on-chip memory to expand physical stream capacity as necessary for a computation to make progress. This creates streams with logical capacities limited only by the total system memory size.

In SCORE, a compute page is a unit of virtualization. Should the entire computation be larger than available hardware, the run-time scheduler time-multiplexes graph compute pages onto physical compute pages. The content of streams between any two pages that are not co-resident in hardware is buffered in special FIFO *stitch* memory segments that the scheduler inserts into the compute graph.

SCORE Hardware. The target hardware is a microprocessor and a reconfigurable array hybrid (Figure 1.1). The microprocessor executes the run-time scheduler and the application control-dominated code. The reconfigurable array executes the fine-grained regular computations in application kernels. The array is a network of Compute Pages (CPs) and Configurable Memory Blocks (CMBs). The run-time scheduler time-multiplexes virtual compute pages on physical CPs and manages buffer allocation in CMBs to create the illusion of unbounded hardware.

The hardware provides streaming connections with back pressure and data presence between the CPs and CMBs, decoupling cycle-by-cycle array timing from the timing of individual pages. Besides facilitating hardware scaling, this relieves the run-time scheduler from the burden of timing closure on the array. The scheduler thus is only responsible for managing coarse grain resources, which include:

- **Compute Pages (CPs)** consist of programmable lookup tables and registers. CPs implement the functionality of application compute graph nodes. A compute node is represented by a CP configuration bit-stream and the contents of its registers and input FIFOs. The scheduler preempts its execution by spilling the state into memory and resumes its execution by restoring the configuration and state to a physical CP.
- **Configurable Memory Blocks (CMBs)** consist of a memory block and a controller that generates addresses and provides a *single* stream port from the memory into the network. Thus while multiple buffers may be allocated in a CMB concurrently, only one can be active at a time. In addition to stream buffers for intermediate computation state, CMBs store CP state and configuration bitstreams. The key benefits of distributed on-chip CMBs include high bandwidth access to application data and the ability to reconfigure many CPs and CMBs in parallel.
- **Off-chip Memory Access Bandwidth** is a scare resource in any integrated circuit due to the limited number and frequency of off-chip IOs. During application execution, input data that resides off chip flows into the array for processing and computed results flow back. Off-chip memory also stores the intermediate computation state that does not fit on chip. The scheduler spills and restores the computation state as necessary at run time.

1.2 SCORE Scheduling

The separation between the compiler and the run-time scheduler in SCORE is similar in purpose to the traditional separation between the function of a compiler and an operating system. The compiler transforms a high-level representation of the computation into a binary form suited for configuring individual hardware resources on the array. The scheduler binds and manages those resources at execution time. This separation delays certain mapping decisions until execution time and adapts those decisions to the specific hardware configuration. Adapting to different device sizes enables application longevity and automatic performance scaling on next generation hardware.

Restricted dataflow models such as synchronous (SDF), boolean controlled (BDF) and integer controlled (IDF) data-flow can in many cases employ a purely static scheduling methodology because they define necessary conditions for graph consistency and bounds on buffering requirements [BML96, Buc93, Buc94]. In contrast, SCORE computation graphs, with nodes that are semantically similar to those in Kahn Process Networks, permit data-dependent token consumption and emission [Kah74]. This results in dynamic flow rates and in general makes it impossible to place static bounds on memory usage. SCORE graphs thus *require* a run-time scheduler to manage stream buffer sizes to guarantee execution correctness.

Run-time dynamic scheduling can often be computationally expensive making application execution inefficient. This is because SCORE scheduling and resource allocation problems are NP-hard. These optimization problems have multiple, simultaneous, independent constraints on memory, communication bandwidth and compute resources. This work presents a quasi-static scheduling methodology that demonstrates that SCORE graphs can be scheduled *efficiently* while retaining the full semantic power and expressiveness of the data-dependent dynamic compute model. This methodology leverages a global rather than greedy view of graph topology and behavior, and therefore yields results superior to those of dynamic schedulers.

In addition to containing the scheduling overhead, the quasi-static methodology identifies the resource management and allocation strategy that contains the overhead of time multiplexing. Time multiplexing requires array reconfigurations and off-chip transfers, both incurring the overheads that must be carefully balanced for application execution to be efficient. An array reconfiguration has a fixed cost proportional to the size of the device. This cost can be amortized by running the scheduled portion of an application for a long period of time after each device reconfiguration. As the application runs, it generates intermediate state that is buffered in special *stitch* segments allocated in on-chip memory blocks (CMBs).

Therefore, the stitch segment capacity, which limits the length of each execution epoch between device reconfigurations, is the key to balancing the two overheads.

The *large* stitch segments enable long execution epochs, which amortize device reconfiguration overhead. At the same time, these large buffers generate significant off-chip traffic every time their contents is temporarily spilled off-chip to bring live state for another part of the computation into limited on-chip memory. In contrast, *small* buffers allow most or all computation state to fit on chip and reduce or completely eliminate off-chip traffic. However, small buffers reduce the length of each execution epoch and do not amortize the overhead of array reconfigurations. Given these competing effects driving buffer sizing, the goal is to find a solution that minimizes the both device configuration and off-chip transfer overheads and therefore the total application execution time.

This work offers a scalable mapping algorithm that adapts to varying memory capacities, reconfiguration overheads, and off-chip bandwidths. The quasi-static scheduler limits the overheads to allow for efficient paged execution.

This report is organized as follows:

Chapter 2: SCORE Scheduling formally defines the scheduling problem and develops an analytical model relating application performance to architecture reconfiguration costs.

Chapter 3: Quasi-Static Scheduling describes in detail the quasi-static scheduler and analyzes implementation trade-offs in SCORE scheduling.

Chapter 4: Temporal Partitioning provides an analytical model relating application performance to reconfigurable array utilization. It continues with several temporal partitioning algorithms and evaluates their results.

Chapter 5: Resource Allocation describes algorithms and implementation details for run-time physical resource management and allocation. The chapter demonstrates the scheduler's adaptability with application performance results.

1.3 Related Work

1.3.1 Multiprocessors

SCORE graph scheduling has obvious similarities with task scheduling for multiprocessors. The reconfigurable array envisioned in this work consists of a network of physical compute pages, analogous to the processors in a massively parallel system. Similar to consumer-producer relationships and synchronization primitives that impose interdependencies on processor tasks, the streams in SCORE compute graphs explicitly specify the precedence constraints between the compute nodes.

Techniques employed for multiprocessor scheduling include priority list scheduling [GDWL92] and gang scheduling [FPR96]. The priority list scheduling is a heuristic employed for minimum latency scheduling with constrained resources. The technique inherently satisfies task precedence constraints while minimizing the total computation time. The operation of the fully dynamic scheduler described in this work is loosely based on priority list scheduling. The dynamic scheduler uses stream token availability to prioritize compute nodes at run-time. However, the static scheduling techniques developed in this work do not rely on priority list scheduling because no run-time information is available to set node priorities. In contrast, the quasi-static scheduler strives to minimize run-time overhead and to move all possible scheduling operations off-line to be performed at application load or install time.

On the other hand, gang scheduling techniques are useful, although apply differently to the SCORE reconfigurable array. Multiprocessor systems have a cost structure that is different from the one of the reconfigurable array. In multiprocessors, the context switch is typically an inexpensive branch instruction, while interprocessor communication is a high latency operation. In contrast, a context switch on a reconfigurable array involves a costly reconfiguration, while the communication between device resources is inexpensive because the latency is hidden in the pipelined interconnect. Gang scheduling for processors involves clustering the frequently communicating tasks on the same processor. This is necessary to take advantage of inexpensive frequent context switches between tasks that communicate through registers and memory, and limit expensive interprocessor transactions. On the reconfigurable array, the frequently communicating compute graph nodes are scheduled on distinct physical compute pages to enable them to compute together for a period of time that amortizes the overhead of array reconfiguration, while taking advantage of low latency communication between CPs and CMBs.

1.3.2 Dataflow Models

The literature offers efficient and near-optimal scheduling solutions for restricted data-flow compute models such as synchronous data-flow (SDF) [BML96]. In SDF the data token input and output rates of individual nodes are static. This permits a compiler to compute specific buffer sizes and verify deadlock free operation prior to execution. SDF scheduling leverages this knowledge of application behavior by computing near optimal schedules statically, thereby completely avoiding any costly run-time overhead.

Less restrictive dataflow models such as boolean controlled (BDF) and integer controlled data-flow (IDF) also define necessary conditions for graph consistency and bounds on buffering requirements [Buc93, Buc94]. Since some of these conditions can be ascertained prior to application execution, the need for dynamic scheduling is reduced or eliminated. In contrast, SCORE dataflow graphs allow for dynamic data dependent token emission and consumption, making it impossible to place a bound on the buffer sizes required for application execution correctness. The run-time scheduler is thus required to guarantee deadlock free operation.

This work applies SDF scheduling techniques to SCORE dynamic dataflow graphs. The primary goal is to dramatically reduce run-time overhead of the fully dynamic scheduler to expand the applicability of SCORE to a wider range of applications. Unlike the more restrictive compute models described above, all scheduling decisions cannot be precomputed in SCORE. However, this work demonstrates that most of the scheduling work can be performed statically without sacrificing SCORE semantics. Furthermore, these static techniques yield a superior application performance when compared to a fully dynamic scheduling approach.

This work applies the SDF scheduling theory to a novel target device — an on-chip network of fine-grain reconfigurable functional units. The concept of actor rate mismatches is modified to develop a temporal partitioning strategy for efficient time-multiplexed application execution. In general, token flow rates cannot be established in SCORE compute graphs, and thus this work precomputes most of the schedule based on the rates obtained from application execution profiles [HL97]. At run-time, a lightweight scheduler manages device reconfiguration and guarantees execution correctness.

Buffer sizing and allocation form a large portion of this work. Maestre *et al* [MKF$^+$01] looked at scheduling of kernels with static rate dataflow in the application critical path on a reconfigurable platform. Their work, however, did not address allocation of variable buffer sizes. This work shows that buffer sizing is a key technique at the scheduler's disposal for controlling the length of application execution epochs and amortizing the overhead of array reconfiguration.

Maestre *et al* dealt primarily with hardware implementations of static rate dataflow graphs. This work includes extensions that allow SCORE scheduling to accommodate applications with multi-rate as well as dynamic rate dataflow. This makes SCORE well suited for applications that include operations such as compression and decompression.

Chapter 2

SCORE Scheduling

This chapter formalizes the scheduling problem and describes the analytical model that relates application performance to reconfigurable array costs.

2.1 Fundamentals

As described in Section 1.1, the SCORE reconfigurable array consists of Compute Pages (CPs) and Configurable Memory Blocks (CMBs), and the computation at the Execution Model level is a graph of virtual compute pages and memory segments. Hardware specifies the size of a compute page to fit on a CP. Two application execution scenarios illustrate the scheduler's role in a SCORE system:

- *Small Design, Large Array* is the simplest scenario. The entire design fits on the array. The number of compute graph nodes is smaller than the number of available physical CPs and CMBs. The scheduler merely maps a compute page graph to selected physical resources once, and configures each physical node to execute until all nodes terminate.

- *Large Design, Small Array* is a common scenario where a design requires more resources than available. In this case the scheduler has a critical role to time-multiplex graph nodes onto physical resources. The scheduler is invoked periodically; it queries the array state, evaluates the computation's progress, and adjusts the set of resident nodes.

The scheduler is also responsible for managing application buffers and resolving an infrequently occurring condition called *bufferlock*. *Bufferlock* may result when a SCORE application that assumes unbounded stream buffers is implemented on a physical array with limited resources (see [Par95] for a formal treatment).

Some applications may require streams to buffer a greater number of tokens than the array stream hardware permits. The failure to provide this buffering capacity results in deadlock. The scheduler monitors the progress made by a design and detects potential deadlock conditions. If the cause is determined to be limited buffering capacity of a stream, the scheduler intervenes to resolve the *bufferlock*. The scheduler "cuts" the offending stream, and a memory block configured as a FIFO is inserted to provide additional buffering. Array execution then resumes. For detailed treatment of *bufferlock* resolution and relevant algorithms, the reader is referred to [Chu00].

2.2 Hardware Implementation and Cost Model

Previously Section 1.1 highlighted the key architectural features of a SCORE reconfigurable array that help facilitate efficient paged execution of applications. They include the following:

Streaming hardware interfaces. Streaming interfaces embedded in the array network make on-chip communication latency insensitive. As a result, the compiler and the scheduler are absolved from resolving the cycle-level array timing and chip-wide timing closure to meet a specified performance target.

Distributed, independently controlled CMBs. The CMBs serve multiple purposes. They offer high bandwidth access to on-chip memory. They store the contents of user-specified memory blocks and *stitch* buffers with intermediate computation state. Additionally, the CMBs cache contexts of compute nodes.

The compute graph node context consists of a CP configuration bitstream and the contents of CP registers and input FIFOs. The bitstream is similar to the program code in a microprocessor system. The register and FIFO contents are akin to stack and heap in traditional programs. To preempt the compute node execution, the scheduler dumps the contents of its registers and FIFOs to a CMB or off-chip memory. To resume the execution, the scheduler configures the node and restores its context. Distributed CMBs enable multiple CPs to be configured in parallel, because the CMBs cache the page configurations and the page states. For example, if a reconfigurable array contains M CMBs, up to M CP could be configured at the same time.

The streaming hardware interfaces and the distributed CMBs are the key features of a SCORE platform that enable the architecture to scale with generations of devices and make *run-time* array reconfiguration affordable. To specify a concrete array implementation, a set of additional architecture parameters must be set. This work assumes the following about the SCORE reconfigurable array.

Reconfigurable Array Controller manages CP and CMB reconfiguration.

Reconfiguration Command	Description
`GETARRAYSTATUS`	request execution status of all CPs and CMBs
`STARTPAGE`	enable a Compute Page (start execution)
`STOPPAGE`	disable a Compute Page
`STARTSEGMENT`	enable a Configurable Memory Block
`STOPSEGMENT`	disable a Configurable Memory Block
`DUMPPAGESTATE`	move compute page state (registers) to a CMB
`DUMPPAGEFIFO`	move compute page FIFO contents to a CMB
`LOADPAGECONFIG`	load compute page configuration bitstream from a CMB
`LOADPAGESTATE`	load compute page state (registers) from a CMB
`LOADPAGEFIFO`	load compute page FIFO contents from a CMB
`GETSEGMENTPOINTERS`	request the contents of CMB address registers
`DUMPSEGMENTFIFO`	move CMB FIFO contents into a CMB memory block
`SETSEGMENTCONFIGPOINTERS`	set the contents of CMB address registers
`CHANGESEGMENTMODE`	specify CMB mode: read, write, read-write
`LOADSEGMENTFIFO`	load CMD FIFO contents from a CMB memory block
`XFERPRIMARYTOCMB`	move a block of off-chip memory into a CMB
`XFERCMBTOPRIMARY`	move CMB contents off chip
`XFERCMBTOCMB`	move CMB contents into another CMB

Table 2.1: Array reconfiguration commands interpreted by the controller.

The scheduler running on the microprocessor sends configuration commands to the controller (see Table 2.1). These commands include basic actions such as dump CP state to a CMB.

This work assumes that the commands arrive one at a time to the array controller. The controller immediately issues a command if there are no resource conflicts, or stalls until the conflict is resolved. A resource conflict occurs when two commands require the same resource to perform their actions. For example, `XFERCMBTOPRIMARY(CMB1,addr)` and `LOADPAGECONFIG(CMB1,CP3)` are conflicting commands, because both require `CMB1`. Any two configuration commands can run in parallel if they do not conflict. Thus, to take advantage of the reconfiguration parallelism offered by the SCORE architecture, the scheduler must distribute compute page contexts evenly across all available CMBs. Furthermore, it must order the reconfiguration commands to minimize resource conflicts.

Single ported Configurable Memory Blocks. A CMB is a block of memory and a controller that manages access to the memory through a streaming interface. While it is feasible to make CMBs with multiple ports, this work assumes that a CMB has a single controller that allows only one data read and one data write port. Although CMB memory could be large enough to fit several user-specified memory segments, *stitch* buffers, and page contexts at the same time, the single controller constraint implies that only one of them can be active at a time.

1:1 CP:CMB ratio. In this work, unless otherwise noted the number of physical Compute Pages equals to the number of Configurable Memory Blocks. Coupled with single ported CMBs, this may inhibit efficient resource utilization in smaller reconfigurable arrays. This is because at high virtualization, computations require large live state storage.

Single off-chip memory port. This work utilizes a single off-chip memory port to transfer application data and results, in addition to page contexts and configurations that do not fit on chip. Because all access to primary memory must be serialized, opportunities for parallel CP reconfiguration are severely reduced.

2.3 Scheduling Problem

This section analyzes the relationship between the total application execution time and the underlying architecture costs. The analysis yields a model to guide temporal partitioning and resource allocation algorithms in the quasi-static scheduler implementation. This section defines the resource allocation problem and discusses the assumptions made in this work and their ramifications on the model and the implementation.

Given the hardware cost model, the scheduler must facilitate the compute graph execution with the goal of minimizing the makespan. With the exception of page firing semantics, the scheduler is ignorant of the contents of compute pages, and it “learns” the application behavior by observing its token flow traffic. To make better scheduling decisions, the scheduler can examine graph topology, run-time buffer fullness, dynamic data token consumption and production rates.

2.3.1 Analytical Model

This section considers a model of application performance on a system with *uniform* buffer sizes and demonstrates system inefficiencies for multi-rate applications. Then the model is extended to accommodate *variable* buffer sizes. Refer to Table 2.2 for the parameters used in the derivations.

Uniform Buffer Model

Consider an application built to process one data token per cycle. A fully spatial implementation will ideally run for K cycles, where K is the number of input tokens, because the compute graph size does not exceed the hardware size. Assuming the N node compute graph is time-multiplexed on a platform with only P

Parameter	Description
B, B_i	absolute buffer size
b_i	minimum *intrinsic* buffer size
C_{reconf}	total reconfig cost per time-slice
C_{array}	array config ovhd per time-slice
C_{swap}	off-chip swap ovhd per time-slice
C_{cp}	cost of configuring a CP
$f(B, L)$	fraction of buffers spilled
K, K_i	num of tokens in a stream
K_{max}	max tokens in a stream ($\max_i(K_i)$)
L	CMB size
M	ave buffer count in a temp partition
N	graph size (compute nodes)
p_i	rate of stream i ($= K_i/K_{max}$)
P	CP count
Q	buffer scaling factor
R	num of reconfigs per execution
S	num of temporal partitions
$s(P, Y)$	config sequentialization param
T_{ideal}	ideal app run time (no overhead)
T_{run}	total application run time
$ts(j)$	the set of streams in time-slice j
V	ratio p_i/b_i for stream i
W_{io}	off-chip bandwidth
Y	CMB count

Table 2.2: A summary of parameters used in the analytical model.

compute pages (CPs), the implementation will ideally run for T_{ideal} cycles:

$$T_{ideal} = KS \tag{2.1}$$

where $S \geq \lceil \frac{N}{P} \rceil$ is the number of temporal partitions.

With the reconfiguration overhead incurred each time-slice, Equation 2.1 becomes $T_{run} = KS + RC_{reconf}$, where R is the total number of array reconfigurations during application execution, and C_{reconf} is the cost of each reconfiguration. T_{ideal}, defined solely by the application and the array size, is the lower-bound on the execution time and cannot be affected by any architecture parameters. To minimize T_{run}, we must address the overhead.

We begin by understanding R, the number of array reconfigurations. Assuming all buffers are of size B and are filled at a constant rate of one token per cycle, then $R = \frac{K}{B}S$. That is, buffers fill up every B cycles, and the system completes only one (K/B)'th of the computation in each iteration through S subgraphs. Substituting

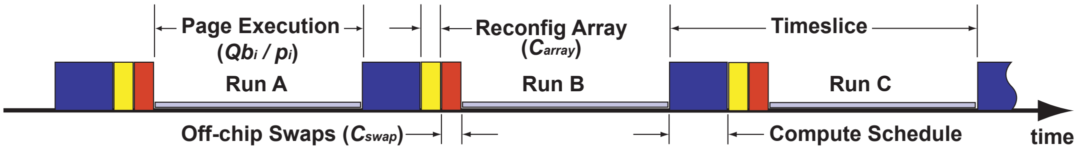


Figure 2.1: Application Execution Timeline.

the expression for R, we obtain:

$$T_{run} = KS + \frac{K}{B} SC_{reconf} \tag{2.2}$$

If Equation 2.2 completely described the relationship between application performance and the hardware costs, simply using large buffers, B, would minimize the execution time. However, the effects on C_{reconf} of finite on-chip memory and limited off-chip IO bandwidth still need to be considered. Array reconfiguration overhead, C_{reconf}, consists of swapping the application state on/off chip and array reconfiguration (Figure 2.1).

$$C_{reconf} = C_{swap} + C_{array} \tag{2.3}$$

Off-chip Transfer Overhead C_{swap} is the cycle cost of spilling and restoring computation state to and from off-chip memory. Assume every temporal graph partition contains M buffers of size B. If the scheduler spills all buffers off chip and loads a new set, the memory transfers consume $C_{swap} = \frac{2MB}{W_{io}}$ cycles with the off-chip bandwidth W_{io}.

Fortunately, not all buffers are spilled. Some reside permanently in the on-chip memories (CMBs), while others are periodically swapped on and off chip. Let L be the CMB size and $f(B, L)$ be a fraction of buffers spilled due to the limited on-chip memory capacity, then the off-chip transfer overhead $C_{swap} = \frac{2MB}{W_{io}} \times f(B, L)$.

The function $f(B, L)$ is highly implementation specific. For example, with a single-port, a CMB contains at most S buffers, one from each temporal partition. The "longest time before reuse" buffer replacement policy is optimal for our cyclical buffer access pattern. Assuming the "longest time before reuse" $f(B, L) = 1 - \lfloor L/B \rfloor / S$. This formulation confirms the intuition that fewer buffers are spilled when CMB size L increases or buffer size B decreases.

Array Reconfiguration Overhead The second component of Equation 2.3 is the overhead of physical compute page reconfiguration. The CMB controllers also require reconfiguration, which includes saving/restoring current address and bounds registers. The cost of configuring a CMB controller is typically negligible in comparison to the CP context switch and will be ignored here for simplicity.

Assuming P compute pages are configured sequentially, the cost is PC_{cp}, where C_{cp} is the CP context switch time. C_{cp} includes dumping CP state, loading a new configuration bitstream, and loading the state of another compute node.

Depending on an architecture and memory allocation, some resources can be configured in parallel. Define a parameter $1 \leq s(P,Y) \leq P$ to represent the scheduler's ability to exploit reconfiguration parallelism in an architecture. If all pages are configured in parallel, then $s(P,Y) = 1$; if all are configured in sequence, then $s(P,Y) = P$. The cost of reconfiguration becomes $C_{array} = s(P,Y) \times C_{cp}$.

In this system, distributed on-chip CMBs enable parallel reconfiguration. If Y CMBs hold page state, up to Y CPs can be configured in parallel, reducing the cost:

$$C_{array} = s(P,Y) \times C_{cp} = \left\lceil \frac{P}{Y} \right\rceil C_{cp} \tag{2.4}$$

Equation 2.4 shows that the entire array can be reconfigured in a single CP configuration time, C_{cp}, if $Y \geq P$. This assumes: (1) CP configurations are evenly distributed among Y CMBs; (2) CP state remains on chip; if CP state is spilled, it is brought on chip at an additional cost.

Complete Model By substituting overhead terms into Equation 2.2, we arrive at the desired model:

$$\begin{aligned} T_{run} &= KS + \frac{KS}{B}\left[C_{swap} + C_{array}\right] \qquad (2.5) \\ &= KS + \frac{KS}{B}\left[\frac{2MB}{W_{io}} f(B,L) + s(P,Y)C_{cp}\right] \end{aligned}$$

Variable Buffer Model

Many applications include multi-rate components with differing input and output stream rates, *e.g.* up/down-sampling; and many applications also include dynamic rate components with varying relative rates, *e.g.* compressors, decompressors. With uniform buffer sizes, B, as previously assumed, multi-rate and dynamic rate applications are guaranteed to utilize resources inefficiently. The assumption that K tokens flow through each stream is not true for multi-rate applications. If K_i is the average number of tokens flowing through stream i, define stream rate $p_i = K_i/K_{max}$, where $K_{max} = \max_j(K_j)$. Note that dynamic rates are modeled as average, static rates, across a range of input datasets. Incorporating multi-rate streams into the model of an ideal execution time, we obtain:

$$T_{ideal} = K_{max} \sum_{1 \leq j \leq S} \max_{i \in ts(j)} (p_i) \tag{2.6}$$

where $ts(j)$ is a set of nodes in time-slice j. Since $0 \leq p_i \leq 1$, the ideal time-multiplexed execution time of a multi-rate application can be lower than that of a single rate application (KS) if high rate and low rate pages are not scheduled together in the same time-slice. The length of a time-slice depends on its own highest rate buffer. Notice that Equation 2.6 is the multi-rate analogue of Equation 2.1. Recall that Equation 2.1 is the ideal execution time of a time-multiplexed computation graph with uniform flow rates. If all stream rates $p_i = 1$, Equation 2.6 reduces to Equation 2.1.

Extending the model for uniform buffer sizes in Equation 2.2 to accommodate multi-rate applications, we obtain:

$$T_{run} = K_{max} \sum_{1 \leq j \leq S} \max_{i \in ts(j)} (p_i) + \left(\frac{K_{max}}{B} \right) SC_{reconf} \tag{2.7}$$

The buffer with the highest data rate (K_{max} tokens) sets the lower bound on the number of times the array is reconfigured. The maximum rate buffer stalls application execution early, even if other buffers still have available space to continue token processing. This memory underutilization can be avoided by setting *relative* buffer sizes B_i to be inversely proportional to the rate of their streams:

$$\forall_{i \neq j} \left(\frac{p_i}{B_i} = \frac{p_j}{B_j} \right) \tag{2.8}$$

This approach, where no single buffer limits application performance, is similar to traditional SDF scheduling [BML96] that allocates token storage proportional to the firing rates of the SDF actors in the minimum balanced schedule.

Although Equation 2.8 constrains the relative buffer sizes, the complete model requires the absolute buffer sizes. If Equation 2.8 holds for all buffers, let b_i be the *intrinsic* buffer size, such that $B_i = Qb_i$ and $\forall_i b_i \geq 1$. While the values of b_i are essentially determined by the application, the scheduler can vary Q, the buffer scaling factor, to obtain absolute buffer sizes B_i that minimize application execution time.

To obtain a model that relates execution time with buffer sizes, replace B with B_i in Equation 2.7, substitute Qb_i for B_i, and then V for the ratio p_i/b_i which is constant for all buffers (Equation 2.8).

$$T_{run} = K_{max} \sum_{1 \leq j \leq S} \left[\max_{i \in ts(j)} (p_i) + \frac{V}{Q} C_{reconf}(j) \right] \tag{2.9}$$

Substituting the actual expression for the $C_{reconf}(j)$ will yield the final model, where $\overline{B}$ is the average buffer size in a CMB.

$$T_{run} = K_{max} \sum_{1 \leq j \leq S} \left\{ \max_{i \in ts(j)} (p_i) + \frac{V}{Q} \left[\sum_{i \in ts(j)} \left[\frac{2Qb_i}{W_{io}} f(\overline{B}, L) \right] + s(P, Y) C_{cp} \right] \right\} \tag{2.10}$$

Here, $f(\overline{B}, L)$ represents a fraction of buffers that on average must be spilled to the off-chip memory in a given time-slice.

Model Review The model confirms our intuition about the relationship between the application execution time and buffer allocation parameters. The first term, T_{ideal} from Equation 2.6, is the lower bound on the application execution time, dependent only on the input size, node rates (p_i), and the array size ($S \geq \lceil \frac{N}{P} \rceil$). The factor $\frac{V}{Q}$ in the overhead term shows that the array is reconfigured less as buffer sizes grow with Q.

The two types of overhead contribute to the total execution time: off-chip memory transfers and array configuration. Off-chip memory transfers increase with Q, while the array configuration overhead decreases with Q. More accurately, the configuration overhead is amortized with Q, which makes time-slices longer.

Consider two outlying scenarios. If off-chip bandwidth W_{io} is high, and reconfiguration overhead dominates, the expression reduces to:

$$T_{run} = T_{ideal} + \frac{SV}{Q} s(P, Y) \times C_{cp} \tag{2.11}$$

which shows that a large Q amortizes reconfiguration overhead and minimizes execution time (Figure 2.2a).

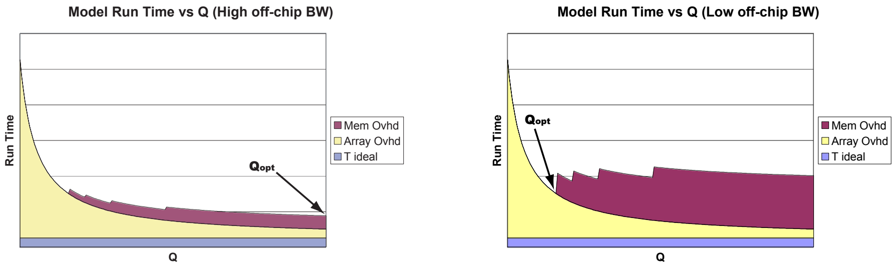
In the opposite case where the off-chip bandwidth is low and memory transfers dominate, the expression reduces to:

$$T_{run} = T_{ideal} + K_{max} \times \sum_{1 \leq j \leq S} \sum_{i \in ts(j)} \frac{2Vb_i}{W_{io}} f(\overline{B}, L) \tag{2.12}$$

suggesting a small Q that eliminates memory traffic and reduces the application execution time (Figure 2.2b). This is because a small Q allows all buffers to fit on chip, reducing average buffer size $\overline{B}$.

2.3.2 Problem Definition

The SCORE scheduling problem that includes the temporal graph partitioning and the physical resource allocation can be posed as follows:



(a) Array overhead dominates: Q_{opt} is high.

(b) Memory transfers dominate: Q_{opt} is low.

Makespan breakdown into off-chip memory transfer overhead ($\propto 1/W_{io}$), array management overhead, and ideal execution time. Q_{opt} is the buffer scaling factor that yields the minimum makespan.

Figure 2.2: Model predicted makespan breakdown.

GIVEN:

- A directed graph $G = (V, E)$
  - A node set $V = C \cup M$ consists of distinct sets of compute nodes C and memory nodes M.
  - Each edge $i \in E$ is annotated with a flow rate p_i.
- Architecture parameters that include CP count N_{cp}, CMB count N_{cmb}, CMB size L, off-chip bandwidth W_{io}, and configuration cost C_{cp}

Dynamic rate SCORE graphs make it impossible to statically determine graph flow rates, p_i's. The model and scheduler implementation rely on *average rates* obtained by profiling an application with a range of input data-sets.

COMPUTE:

with the goal to minimize T_{run} (Equation 2.10)

1. Nonoverlapping partition $\mathcal{P} = (P_1, P_2, ..., P_S)$ of the node set V, that meets resource constraints:

$$\forall_{j \in [1,S]} \, |P_j \cap C| \leq N_{cp} \tag{2.13}$$

$$\forall_{j \in [1,S]} \, |P_j \cap M| + |T_j| \leq N_{cmb} \tag{2.14}$$

where T_j is the set of edges entering or leaving P_j:

$$T_j = \{(n, m) \in E \mid \forall_{i \neq j} (n \in P_i \wedge m \notin P_j) \vee (n \notin P_i \wedge m \in P_j)\}$$

The set T_j contains the *stitch* buffers that are inserted between temporal partitions to buffer intermediate computation state and consume CMBs.

2. Stitch buffer sizes, consisting of the *intrinsic* buffer sizes $b_i \geq 1$ and the scaling factor $Q \geq 1$, such that

$$\bigwedge_{\substack{i,j \in E \\ i \neq j}} \left(\frac{p_i}{b_i} = \frac{p_j}{b_j} \right) \tag{2.15}$$

and $Qb_i \leq L$, where L is the CMB size.

3. Assignment $R_p[]$ of compute pages to CPs:

$$\forall_{n \in C} R_p[n] \in [1, N_{cp}] \tag{2.16}$$

such that no two compute pages in a partition occupy the same resource:

$$\forall_{j \in [1,S]} \forall_{\substack{n,m \in P_j \\ n \neq m}} R_p[n] \neq R_p[m] \tag{2.17}$$

The entities assigned to a CMB include user memory segments (set M), inter-partition stitch buffers (set $T = \cup_j T_j$), and compute page state buffers (one for each page in C). Let $B = M \cup T \cup C$. Compute the assignment $R_m[]$ of members of B to CMBs:

$$\forall_{i \in B} R_m[i] \in [1, N_{cmb}] \times (A_{start}..A_{end})_i \tag{2.18}$$

where the buffer size is

$$A^i_{end} - A^i_{start} = \begin{cases} B_i & \text{if } i \in M \\ Qb_i & \text{if } i \in T \\ k_{size} & \text{if } i \in C \end{cases} \tag{2.19}$$

and such that no two memory segments in a partition occupy the same resource:

$$\forall_{j \in [1,S]} \forall_{\substack{n,m \in P_j \\ n \neq m}} R_m[n] \neq R_m[m] \tag{2.20}$$

Each mapping $R_m[n] = k_n \times (A_{start}..A_{end})_n$ contains the CMB identifier k_n and the address range $(A_{start}..A_{end})_n$ occupied by the buffer n. No two segments in a partition can share a single-ported CMB:

$$\forall_{j \in [1,S]} \forall_{\substack{n,m \in P_j \\ n,m \in M \cup T \\ n \neq m}} R_m[n].k \neq R_m[m].k \tag{2.21}$$

In the problem statement, we have made several assumptions, that can be relaxed at the expense of added complexity to the allocation algorithms.

- Buffers do not share space dynamically. Conceivably, since a running application depletes source buffers and fills sink buffers with data, a source buffer and a sink buffer could share physical space. However, since the locally dynamic rates can differ from the measured, long term averages, one cannot exploit this kind of buffer sharing without considerable additional memory management complexity.

- Buffer space is allocated based on the computed buffer size when the buffer is full. The system cannot allocate the same address space for two buffers from different temporal partitions, unless the contents are spilled between time-slices. For example, a buffer depleted in time-slice 1 and filled in time-slice 4 cannot share the same space with a buffer filled in 2 and depleted in 3, because the degree to which a buffer is depleted may vary depending on an input data-set.

- Buffer spills are *atomic*. The entire buffer content is forced off-chip to free space for another buffer, even if only a fraction of that space is required.

The analytical model and the problem statement presented in this section serve as the guideline for algorithm development for the quasi-static scheduler system. The following chapters elaborate on the implementation details to demonstrate the system's ability to adapt to variations in reconfigurable array parameters. These variations encompass changes in the underlying hardware costs and available resources.

Chapter 3

Quasi-Static Scheduling

This chapter answers the question, “How does one choose the system implementation that both yields high quality schedules and incurs low, manageable overhead?” It describes the space of scheduler implementations and evaluates several points in that space.

3.1 Space of Scheduling Solutions

The problem of low run-time overhead scheduling of data-flow graphs has been solved for some restricted data-flow models that include synchronous data-flow (SDF) [BML96]. Under SDF, for example, the data token input and output rates of individual nodes are static. SDF scheduling leverages this knowledge of application behavior by computing near optimal schedules off-line, completely avoiding run-time overhead. A static schedule is simply a sequence of actor firings.

More expressive models such as boolean (BDF) and integer data-flow (IDF) also offer opportunities for mostly static scheduling [Buc93, Buc94]. In addition to static rate actors of SDF, these models offer explicit control actors such as multiplexors and switches. For non-terminating, deadlock-free execution with bounded buffer memory, token flow rates must meet certain consistency conditions. These conditions often depend on the actual boolean or integer values emitted by actors at run-time. Static analysis and knowledge of actor semantics may permit a compiler to ascertain off-line whether a graph meets these conditions. A strongly consistent graph, that meets all consistency conditions with any input data set, can benefit from low overhead, purely static scheduling. However, a weakly consistent graph, that meets consistency condition only with some input data, requires greater scheduler involvement at run-time. In such a case, a *quasi-static* schedule can be generated to reduce run-time decision making to a minimum. A *quasi-static* schedule describes execution of a boolean or an integer data-flow graph and consists of a sequence of actor firings. Some firings are predicated on the token values emitted by actors during graph execution.

Heterochronous data-flow (HDF) is another compute model that is related to SCORE and can benefit from static scheduling [GLL99]. Both SCORE and HDF graph nodes contain a finite-state machines that control computation data-paths, which are effectively SDF components. Unlike SCORE, to permit static analysis, HDF restricts state transitions to only occur between schedule iterations. Thus for each combination of actor states, HDF graph behaves as a SDF graph with a unique type signature of token flow rates that can be used for static analysis. The questions of deadlock and buffer bounds are decidable off-line. However, static scheduling is not always practical since the number of states in the compute graph is exponential in the number of actors. In such a case, some scheduling decisions may be postponed until run time.

In the restricted compute models discussed above, a scheduler takes advantage of known graph node behavior to reduce run-time overhead. In SDF, actor firing rates are static. In BDF and IDF, the static rate components are handled as SDF graphs, while the known semantics of control actors combined with observable run-time values provide an opportunity for static analysis and generation of a quasi-static schedule. An HDF graph is simply a set of SDF graphs, and each synchronous graph can be verified to be consistent. Restrictions in these models allow for low overhead scheduling, and in some cases provable deadlock free execution with bounded buffers.

In contrast, SCORE compute graphs offer greater expressibility to a programmer, but no information about compute node semantics to a scheduler. Given the intended hardware target, run-time observability of the data emitted by graph nodes is very limited. A scheduler can make no *a priori* assumptions about application behavior, but must observe computation progress to make scheduling decisions. A dynamic scheduler seems to be a natural fit for SCORE graphs because it can handle dynamic data-driven application behavior. However, it comes at the cost of high run-time overhead.

Notice that both extremes of fully dynamic and fully static scheduling perform the same basic set of operations as do all implementations in between. All schedulers compute node firing sequence and timing, and allocate physical resources to nodes and communication links. What separates these approaches is the time when scheduling decisions are made. We must recognize these extremes and the space between them to understand the opportunities that exist for low overhead, high quality scheduling for SCORE.

In [Lee91], Lee forms a taxonomy of scheduling solutions and explores the space between fully static and fully dynamic approaches. The author attempts to

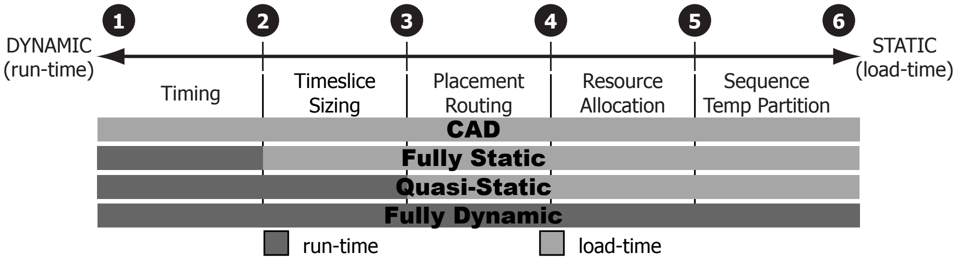


Figure 3.1: Space of scheduling solutions, which includes (1) FPGA CAD, (2) a fully static (*e.g.* SDF), and (6) a fully dynamic scheduler (*e.g.* Section 3.3). Quasi-Static scheduler (3) is discussed in Section 3.4.

find a compromise between a low overhead static and a high overhead dynamic scheduling for applications with data-dependent data-flow. Lee demonstrates efficient hybrid scheduling techniques that employ dynamic scheduling only when absolutely required. For data-flow models such as BDF and IDF, although it is impossible to deterministically optimize the statically computed schedules, good compile-time decisions frequently remove the need for dynamic scheduling or load balancing [HL97, Ha92, HL91].

A goal of this work is to expand on the taxonomy in [Lee91] by identifying its analogue for SCORE. To run an application on a reconfigurable array, the scheduler must perform five specific inter-dependent steps shown on Figure 3.1. One way to represent a spectrum of SCORE run-time resource management solutions is as a one-dimensional space of arranged scheduling steps. Each point represents a scheduling solution and cuts the space into two parts: steps performed dynamically and steps performed statically. For example, in Figure 3.1 point 3 represents a scheduling solution where steps to the left (*Timing* and *Timeslice Sizing*) are performed dynamically, *i.e.* at run-time. Steps to the right (*Place/Route*, *Resource Alloc*, and *Sequence/Temp Partition*) are performed statically, *i.e.* at application load/install time.

Figure 3.1 shows six possible scheduler implementations that differ in run-time complexity and overhead as well as scheduling optimality. The boundaries between these steps are not rigid due to close interdependence between operations. Nevertheless, consider this diagram to represent feasible implementations of run-time resource management solutions for SCORE.

Let us look at each scheduling step in detail.

- *Sequence/Temporal Partitioning* partitions the graph into a sequence of precedence constrained, schedulable sub-graphs. A schedulable subgraph is one that “fits” on the array. This means that each virtual page requires a physical CP, each user memory segment requires a CMB, and each stream crossing a temporal partition boundary must be buffered by a CMB. While computing the sequence is generally a straight-forward process constrained only by graph topology, temporal partitioning with optimization(s) is an NP-hard problem. Optimizations include minimizing buffering requirements, maximizing hardware utilization, minimizing cuts in graph cycles to prevent thrashing, and maximizing temporal locality of intermediate data.

- *Resource Allocation* maps the schedulable subgraphs down to physical resources in an "ideal array" without routing constraints. Virtual pages are assigned to physical CPs; virtual memory segments are assigned to CMBs, and memory is allocated in the assigned CMBs. This step primarily attempts to maximize on-chip CMB memory utilization in an effort to reduce transactions with slower primary memory.
- *Placement/Routing* maps the nodes of the "ideal array" onto the same size real array with a network that constrains routing. This step may fail if the scheduler made wrong assumptions about the array's routing structure or its physical layout in two previous steps. Should routing or placement fail, the scheduler must return to *Temporal Partitioning* and repeat the first two steps with tighter constraints.
- *Timeslice Sizing* computes a time interval for each schedulable subgraph to be resident on the array. This step closely depends on *Temporal Partitioning*, allocated buffer sizes, and I/O token rates intrinsic to individual nodes.
- *Timing* is responsible for cycle-by-cycle operation of the array hardware. Software tools such as those in FPGA CAD flows (point `1`) compute conservative timing statically for FSMs, data-paths, and communication components in a design. However, the SCORE scheduler relies on the array hardware to support dynamic timing using network interfaces with flow control and the ability to stall compute pages (CPs) and configurable memory blocks (CMBs).

  Flexible timing enhances the model's robustness to target device changes, enabling a scheduler to manage resources on an array of any size and/or family. This is true as long as common reconfiguration commands are supported and data integrity is guaranteed by the communication protocol. Contrast this with existing FPGA CAD tools that severely limit design scalability and compatibility among target devices.

Independent of implementation details, every SCORE run-time scheduler is responsible for each of the steps above. Figure 3.1 marks `1` through `6` as clear places where cuts, that divide the space into dynamic and static sub-spaces, can turn into implementations. Some steps, such as *Timing* may be implemented efficiently in array hardware, thus obviating direct involvement of the run-time software scheduler. Although the proper balance of efficiency, functionality and flexibility is difficult if not impossible to attain, below we summarize the key issues driving the selection of a specific solution based on the underlying implementation requirements.

- **Run-Time overhead** is the most obvious. Although the scheduler at `1` incurs no run-time overhead, the overhead gradually increases as the scheduler implementation performs more steps at run-time. Two factors determine the run-time overhead: the scheduling algorithm complexity and the frequency at which the run-time system checks the application status.

  Scheduling algorithm complexity is another factor that determines the overhead. A simple scheduling algorithm yields a low overhead implementation, but sacrifices the application performance even in the *ideal* case, where no overheads exist.

  The frequency at which the run-time system checks the application status determines how stale the information on which the scheduler bases its decisions. In theory, to attain the highest scheduling quality, the system must be aware of the exact application status at every clock cycle of its execution. Cycle accurate information such as stream buffer fullness and the nodes starving due to insufficient input tokens would result in a good schedule, at the cost of very high run-time overhead.

- **Scheduling quality** is highly dependent on the scheduler's knowledge of application behavior, predicted and observed. For schedulers performing the majority of steps statically, accurate prediction of application behavior is critical for schedule quality. For example, in SDF, close-to-optimal schedules can be constructed statically. As a scheduler moves from `2` toward `6` and performs more steps at run-time, application behavior can be predicted and monitored. For every scheduler from `2` to `6`, both *accuracy* of predicted and *currency* of observed information determine scheduling quality. There is a trade-off between the age of an observation and the overhead of collecting it. Presumably, if array state is monitored continuously, a dynamic scheduler would be a superior if costly solution.

- **Advanced SCORE features.** SCORE permits instantiation of graph nodes and creation of subgraphs at run-time, allowing a computation to be composed dynamically, and share the reconfigurable array with other applications. These features require *Resource Allocation* and *Temporal Partitioning* to map virtual to physical resources at run-time. Although this seems to imply that a fully dynamic scheduler is necessary, a static scheduler may be more efficient if the changes to graph topology are infrequent. The static scheduler simply has to recompute the schedule to reflect the changes.

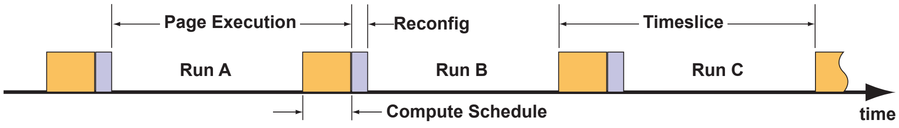


Figure 3.2: Application Execution Timeline.

Although every scheduling solution in the implementation space discussed above can implement the complete SCORE semantics, only a subset performs efficiently. Independent of the actual implementation, a great variety of control-dominated computations can be expressed without restrictions (*e.g.* compression, sorting, selection). This allows efficient exploitation of the powerful semantics of SCORE in a low overhead scheduling implementation.

3.2 Scheduling Overhead and Timeslice Size

Consider the impact of the scheduling overhead on the application performance. How critical is the run-time overhead as a factor guiding the selection of the scheduler implementation? The application execution timeline (Figure 3.2) shows that scheduling and reconfiguration overheads form the lower bound on the time-slice size in the system developed in this work. Time-slice size determines the ability of a run-time scheduler to react to changes in application behavior. That includes the decision to remove compute pages that cannot proceed with execution (*e.g.* out of input tokens) and to bring in the new ones that can run.

Although this work evaluates a time-slice based execution model, the scheduling overheads affects event-based models similarly. Consider a system where a page or a group of pages signals to the scheduler when they cannot make further progress. The scheduler must remove the stalled nodes, compute a new set of nodes to map to resources that became available, and reconfigure a part of the array. Large overhead restricts the response time of the scheduler, and thus limits applicability of SCORE to some applications. Consider several scenarios, where high overhead leads to a very inefficient system:

- The performance of applications with short total execution time is dominated by the run-time overhead of scheduling for the first time-slice.

- When the number of pages in a closed feedback loop is larger than the number of physical pages, the amount of useful computation per time-slice will be limited by the number of tokens in the feedback loop. If this number of

> tokens is small compared to the reconfiguration and scheduling time, then reconfiguration and scheduling overhead will dominate the useful computing time. This is a phenomenon similar to virtual memory thrashing that occurs when the working set does not fit into available physical memory.

Clearly, to broaden applicability of SCORE, scheduling must incur low overhead, while providing acceptable results. The following two sections examine several points in the space of scheduler implementations to evaluate their performance and the overhead.

3.3 Fully Dynamic Scheduler

Because few restrictions exist on token flow behavior in SCORE compute graph nodes, a fully dynamic run-time scheduler, point 6 on Figure 3.1, was a natural first choice for the the system implementation. The scheduler is designed to handle large data-dependent variations in page token consumption and emission rates, and therefore it makes decisions driven largely by token availability at page inputs. A dynamic scheduler continuously monitors active computation progress on the array and adapts the schedule to match the observed application data-flow patterns. The resulting schedule quality depends heavily on the temporal *granularity* of monitoring and scheduling decisions. For instance, fine-grained, cycle-by-cycle scheduling may result in near-optimal schedules but incurs prohibitively expensive run-time overhead.

3.3.1 Functionality

The dynamic scheduler implementation was primarily an effort by Michael Chu. For a complete description and analysis, see [Chu00]. This section discusses the dynamic scheduler operation as it is used as a point of reference for the Quasi-Static scheduler developed in this work.

The dynamic scheduler uses a version of *priority-list scheduling*. Instead of evaluating all graph nodes as candidates using a priority function, only the nodes whose predecessors have been or are scheduled and thus satisfy the precedence constraints are considered. These candidate nodes form a “frontier” that moves downstream across a compute graph. The priority of a candidate is directly proportional to the availability of input tokens and output space. Alternatively, the algorithm can be thought of as a greedy, breadth-first packing of nodes onto the array.

Figure 3.3 demonstrates the relationship between dynamic scheduler modules. The scheduler is invoked in each time-slice to perform the following sequence of

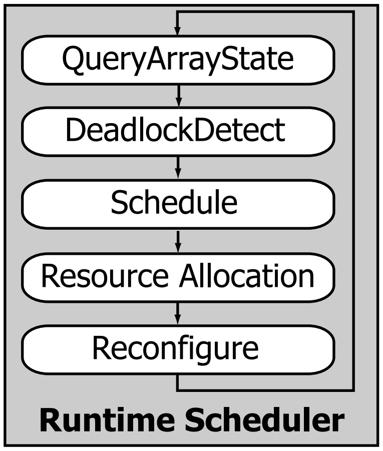


Figure 3.3: Fully Dynamic Scheduler: module flow in the critical loop.

operations. Each timeslice length is fixed to 250,000 cycles. On the application execution timeline (Figure 3.2) these operations are collectively marked as "Compute Schedule":

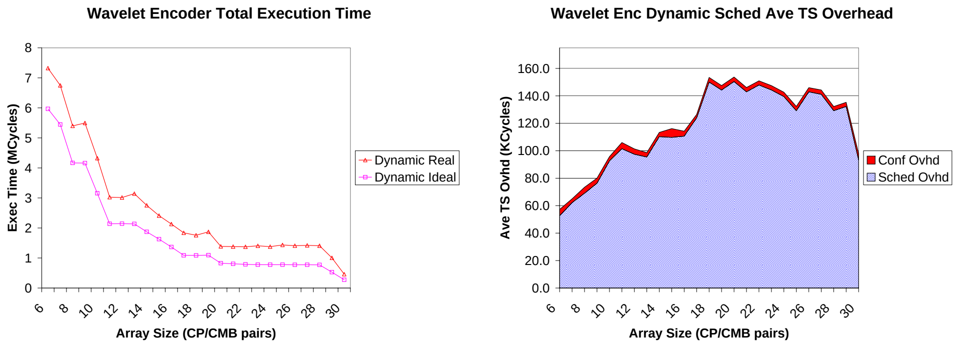
- *Query Array State* obtains execution statistics from the array hardware, updates corresponding scheduler data structures. It also identifies pages to be removed, including pages that terminated or exhibited low firing activity.

- *Deadlock Detect* verifies that a resident computation is making progress. Failure to detect reasonable progress on the array forces the scheduler to invoke deadlock detection and resolution algorithms. Bufferlocks are resolved as described in Section 2.1, and deadlocked processes are killed.

- *Schedule* identifies the array resources that became available after the scheduler removed low activity and terminated nodes. This module then attempts to pack the array with the nodes from the "frontier" priority list, which are expected to make progress if scheduled. The scheduler inserts special *stitch* segments to buffer the contents of streams crossing temporal partitions. This module outputs a page subgraph that is guaranteed to fit on the array, which will be scheduled in the subsequent time-slice.

- *Resource Allocation* assigns the subgraph nodes to physical compute pages (CPs) and memory blocks (CMBs).

- *Reconfigure* issues a sequence of commands to the array controller to load



(a) Application Runtime

(b) Ave Time-slice Overhead

Figure 3.4: Wavelet Encoder (30 pages) performance summary for the Fully Dynamic Scheduler.

the subset of nodes selected by the previous modules and to resume execution.

3.3.2 Analysis of Performance

Figure 3.4(a) shows total execution time for one SCORE application, a wavelet-based image encoder that requires 30 physical pages for a fully spatial implementation. The vertical axis shows the total execution time to encode a 512 × 512 bitmap. The horizontal axis shows the array size in compute page and configurable memory block pairs. Performance was measured on a cycle-level array simulator. The two curves shown on the diagram represent the execution times — one on a simulated realistic system and the other on a simulated idealized system. The idealized system does not injure the run-time computational overhead of the scheduler. Both curves exhibit expected performance scaling behavior. In general, more hardware results in an equal or lower execution time. However, this trend is not strictly monotonic in the hardware size due to anomalous effects in the dynamic scheduler.

The scheduler implementation was heavily optimized. In attempt to reduce run-time overhead while maintaining schedule quality, all non-essential components were eliminated, and remaining code was redesigned to improve memory allocation and layout of data-structures. As shown on Figure 3.4(b), the average run-time scheduling overhead ranges from 50 to 150 thousand cycles per timeslice. Compare that to a small overhead of less than 10 thousand cycles for array reconfiguration. This high scheduling overhead translates into 35% of the total application execution time. Similar high run-time overhead is observed with other applications. Refer to Appendix B.1 for complete results.

Having no basis for comparison, little can be concluded even in an idealized system about the scheduling quality. Clearly a more complex algorithm than the one implemented would be required to further improve application performance. With a high run-time overhead, an attempt to improve the run-time scheduler quality may further constrain the set of practical applications for SCORE.

The dynamic scheduler may provide acceptable results for applications with unlimited total execution time or a very large number of execution cycles on the order of 100,000s. However, as discussed in the previous section, high scheduling overhead may make a SCORE implementation inefficient. Since the micro-architectural design exhibits array reconfiguration time on the order of only 10,000 cycles or less, dynamic scheduling becomes the primary bottleneck preventing efficient execution of a range of applications.

3.4 Quasi-Static Scheduler

A viable and efficient alternative to the Fully Dynamic scheduler, point 6, is the quasi-static scheduler that corresponds to point 3 on Figure 3.1. The quasi-static scheduler computes graph temporal partitioning, execution sequence, and resource assignment off-line. Timeslice sizing is the only operation performed at run time. The quasi-static approach adapts to the slowly varying application run-time characteristics, because it recomputes a high quality, static schedule very infrequently to reflect only substantial changes in the compute graph topology or page firing rates.

3.4.1 Basic Implementation Principles

Static Scheduling of Dynamic Dataflow Graphs

A Fully Dynamic Scheduler makes its decisions based on the graph topology and the run-time token availability of individual compute pages. In contrast, the quasi-static scheduler uses the graph topology and predicted page token emission and consumption rates. Although in SCORE the token rates are dynamic and data-dependent in general, many practical applications such as image codecs share the following common characteristic. While their instantaneous, short-term token emission and consumption rates are dynamic and rapidly varying, their long-term behavior is on average static and bounded.

Consider, for example, a Huffman encoder in the JPEG encoder application. In theory, the output data rate cannot be bounded. In practice, measurements performed in this work show that the token emission rate has tight lower and upper bounds over a large range of images.

A scheduling system that uses the average, long-term token production and consumption rates for its decisions, can operate effectively and correctly as long as it can handle the rare cases when the token production rates greatly exceed the expected average. The rates used in the quasi-static scheduler are the averages obtained by profiling the previous executions of an application. Additional information can also be obtained from profiling, including the standard deviation on rate distribution for each individual operator.

Relationship to SDF scheduling

The quasi-static scheduler expands on existing work to map synchronous data-flow (SDF) programs to uni- and multi-processors [BML96]. An SDF program is a data-flow graph whose computational nodes (*actors*) communicate via *arcs* using the same streaming discipline as SCORE. This work adapts SDF analysis and algorithms for SCORE by equating a page to an actor.

SDF is well established in the literature and has been successfully used to map signal processing algorithms to a variety of processor platforms. However, SCORE scheduling for a hybrid reconfigurable architecture differs in two key ways from scheduling SDF on microprocessors.

First, SDF actors are restricted to having static input/output rates. An example of SDF is an adder repeatedly consumes two inputs and produces one output. In contrast, SCORE pages may have dynamic input/output rates. An example of a dynamic page is a Huffman encoder. Dynamic rates make it impossible to determine a static bound on the run-time requirements for buffer memory. In the absence of such a bound, the SCORE scheduler computes a quasi-static schedule from *average* input/output rates and makes an allowance at run-time for expanding stream buffers and modifying the schedule.

Second, the SCORE reconfigurable architecture has different execution costs than a microprocessor running SDF and hence requires different approaches for optimizing the schedule. An SDF program for a uni-/multi-processor target is typically scheduled at compile time as a collection of threads, one per microprocessor. Each thread repeatedly evaluates a subset of actors.

In processor systems, the cost of inter-processor communication is higher than memory access, consuming 10s to 100s of cycles. Therefore, an SDF schedule optimizes for minimum local buffer sizes by evaluating each actor a minimum number of times in turn. The cost of switching to evaluate a different actor is low, potentially as inexpensive as incrementing the program counter or taking a branch, typically several clock cycles. Because frequent actor switching is acceptable, SDF techniques tend to cluster *communicating actors in the same processor*, running them time-multiplexed with memory-buffered communication.

On reconfigurable hardware, however, switching to evaluate a different page is expensive, requiring saving and restoring a page context, typically hundreds to thousands of clock cycles. Consequently, a SCORE schedule prefers to reuse a page for many consecutive cycles to amortize the cost of reconfiguration. On the other hand, the reconfigurable hardware makes inter-page communication primitives cheap, ideally offering single-cycle pipelined send/receive. Consequently, SCORE takes advantage of spatial concurrency enabled by a network of compute pages (CPs) and schedules *communicating actors on separate CPs* to reduce the number of context switches.

Minimization of context switches has also been addressed in SDF scheduling with a technique known as optimal vectorization. Vectorization can help generate minimum activation schedules for scalable synchronous dataflow (SSDF) graphs [RPZM93]. In each activation, a scalable SDF graph can consume and produce any integer multiple of the actual rates defined by its actors. An activation is a series of operations to begin executing a (sub)graph; in a multi-threaded implementation, an activation is a context switch. The integer multiple of the rates is known as the blocking factor. Optimal vectorization is a transformation on a SSDF subgraph that increases by a blocking factor the number of tokens consumed and produced per (sub)graph activation. This transformation attempts to minimize the number of block activations while meeting hardware resource constraints. Notice that vectorization increases stream buffering requirements to improve application performance.

Scheduling of SCORE compute graphs is related to scheduling of scalable synchronous dataflow (SSDF) graphs [RPZM93]. The scheduler executes each SCORE subgraph for a period of time that amortizes the high overhead of context switches. Execution length depends directly on the amount of on-chip memory available to buffer intermediate results. The buffer scaling factor Q, defined in Section 2.3.1, determines the amount of available space for intermediate computation state. This scaling factor is analogous to a blocking factor in SSDF, which must be carefully selected to minimize activations (context switches) and meet hardware imposed constraints. Section 5.2.1 discusses an algorithm to select the optimal buffer scaling factor Q.

Execution Correctness

The quasi-static scheduler produces *single-appearance* schedules based on application compute graph topology and token emission and consumption rates of compute pages. Each compute graph node appears in the schedule exactly once, and therefore the quasi-static scheduler provides the same computational semantics as the dynamic scheduler.

In general, time-multiplexed execution using physically bounded buffers must produce exactly the same *functional* result as would a fully spatial implementation with unbounded buffers. To see why this is true, note the following:

- The behavior of a SCORE graph is completely deterministic and independent of operator timing. This is a consequence of the execution semantics that an operator can fire only when input data tokens are available. Hence, pages can be scheduled in any order without changing the semantics of the graph.
- A schedule that includes every virtual page gives every page an opportunity to fire on each schedule iteration.
- The quasi-static scheduler iterates through its schedule until all pages have completed. Hence, regardless of the order in the schedule, every page has an opportunity to consume all of its inputs and produce all of its results.
- As long as the array is not deadlocked, the virtual graph makes computational progress on every schedule iteration.
- Should bufferlock occur, the scheduler expands the full buffers to provide the illusion of unbounded buffers up to the available memory in the system.

Therefore, any graph executing without a deadlock on unbounded hardware, does not deadlock when time-multiplexed onto limited physical resources by the quasi-static scheduler. The quasi-static schedule produces the same functional results as the unbounded case. Note that a deadlock occurs only if the application's total buffering requirements exceed the physical system memory; however, this is no different from an application running out of memory in a conventional processor system.

3.4.2 Hardware Support

Timeslice Sizing step separates a fully static scheduler from the quasi-static scheduler, points 2 and 3 in Figure 3.1. *Timeslice Sizing* determines the length of time each scheduled subgraph resides on the array. While the fully static scheduler specifies *a priori* precise periods of time to schedule each subgraph, the quasi-static scheduler implemented in this work relies on special array hardware to determine the size of each time-slice. The hardware detects stall conditions on the array and triggers the run-time reconfiguration engine to go to the next time-slice.

Stall conditions, which include *empty* input buffers and *full* output buffers, impede any progress of the resident subgraph. The reconfiguration script includes commands to configure CMBs to detect these conditions. When a stall condition occurs, the array controller interrupts the processor to invoke the run-time array reconfiguration engine. The engine then configures the array for the next subgraph in the pre-computed schedule. Experiments have shown this simple mechanism to be effective and inexpensive. A typical subgraph may run anywhere from ten thousand to one hundred thousand cycles, hence cycle-precise interrupts are not required from the stall detection. A small latency of 10–100 cycles in reporting can easily be tolerated, permitting a simple hardware implementation.

The stall detect hardware allows the scheduler to adapt gracefully to dynamic token emission and consumption rates of compute pages and the processor. The processor executes control-dominated parts of an application and effectively acts as a very dynamic, unpredictable compute node that exchanges data tokens with the part of the computation running on the reconfigurable array.

The fully dynamic scheduler in Section 3.3 uses a fixed size time-slice for the *Timeslice Sizing* step and analyzes page activity in *Sequence* and *Resource Allocation* at run-time. In contrast, the quasi-static scheduler with hardware *stall detect* analyzes CP activity in *Timeslice Sizing* as the application runs and computes the schedule at load-time by predicting application behavior from its graph topology and profiled rates. Furthermore, with stall detect, the quasi-static scheduler uses *buffer sizing* to control subgraph execution time between scheduler actions, *i.e.* to actively vary time-slice size to manage array reconfiguration overhead as Section 2.3.1 describes.

3.4.3 Quasi-Static Scheduler Implementation

General System Tool Flow

The quasi-static scheduler consists of a static schedule generator and a run-time reconfiguration engine as shown in Figure 3.5. The modules of the quasi-static scheduler are similar to those of the dynamic scheduler in Figure 3.3. Rather than running in the critical scheduling loop, they are factored into two components such that certain tasks are performed less frequently than every time-slice. The scheduler inner loop, *i.e.* the work that incurs run-time overhead at every time slice, is thus substantially reduced.

The infrequently running component is the schedule generator. It analyzes the virtual page graph plus the profile information from previous application runs to produce a schedule in the form of a script of array reconfiguration commands. The frequently running component, the inner loop, is the reconfiguration engine, which

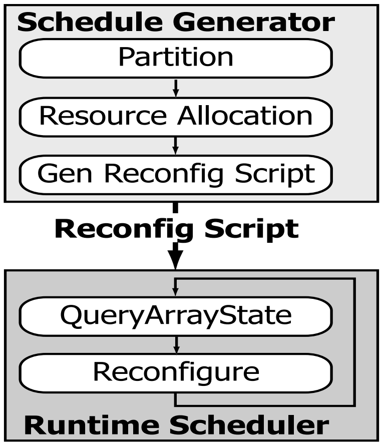


Figure 3.5: Quasi-Static Scheduler: module flow in static and run-time components.

oversees graph execution by issuing script commands to the hardware. Particular differences from the dynamic scheduler are as follows:

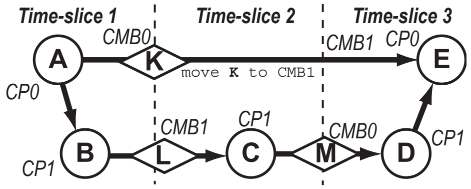
- *Query Array State* detects only the "done" signals from nodes.
- *Deadlock* detection and resolution are not implemented, keeping the system flow simple for experiments with resource allocation algorithms. Note, however, that deadlock detection remains inexpensive in the quasi-static scheduler, requiring only page activity counters in hardware and minimal house-keeping in software. Resolving a *bufferlock*, which is infrequent, is expensive and typically requires regenerating a schedule.
- *Schedule* and *Resource Allocation* are performed by the schedule generator.
- *Reconfigure* is replaced by a small configuration script execution engine.

By simplifying or eliminating most of run-time components, the quasi-static scheduler incurs on average only *one eighth* of the per-time-slice run-time overhead of the fully dynamic implementation. The following section discusses more comprehensive results.

Static Schedule Generator

The static schedule generator analyzes the compute graph and the application execution profile to compute an array reconfiguration script. The *Partition* module performs temporal partitioning of the compute graph and computes the precedence constrained node sequence. *Resource Allocation* maps the virtual compute nodes onto array hardware and allocates CMB memory for *stitch* segments and

	Time-slice		
Array Resource	1	2	3
CP0	A		E
CP1	B	C	D
CMB0	K[0:10]	M[10:20]	M[10:20]
CMB1	L[0:20]	L[0:20]	K[20:30]



The compute graph above is temporally partitioned to run in 3 time-slices. A static schedule table shows ordered virtual nodes assigned to array resources. Each row corresponds to a compute page (CP) or a configurable memory block (CMB), and each column represents a time step. Segments are annotated with their locations in CMBs.

Figure 3.6: An example of a static schedule.

user-specified memory blocks. Chapters 4 and 5 analyze these modules in detail. The last module *Generate Reconfiguration Script* emits array reconfiguration commands arranged to maximize parallel configuration of nodes on the array.

The static schedule generator produces a schedule describable as a table containing a fixed sequence of resource mappings shown in Figure 3.6. Each column represents a time-slice. Each row corresponds to a physical array component, namely a compute page (CP) or configurable memory block (CMB). The table also contains a CMB memory location for each segment. For simplicity, the table does not show the CMB locations that store the the register state, input FIFO contents and configuration bit-streams for pages that are inactive. Converting the schedule table into a sequence of array reconfiguration actions requires additional analysis, especially to take advantage of SCORE array capability to reconfigure multiple CPs in parallel. Refer to Section 2.2 for details.

To simplify the job of the run-time reconfiguration engine, the schedule emitted by the static schedule generator is a script of reconfiguration commands, instead of a table. The commands are arranged to minimize resource conflicts to reconfigure as many CPs as possible in parallel. Because all scheduling decisions with the exception of the timeslice length are precomputed, the script minimizes the runtime processing.

```
 17 XFERPRIMARYTOCMB [addr = 1] ----> CMB0[addr = 21680]
 18 XFERPRIMARYTOCMB [addr = 0] ----> CMB1[addr = 0]
 19 XFERPRIMARYTOCMB [addr = 3] ----> CMB1[addr = 62640]
 20 LOADPAGECONFIG CMB1[addr = 0] ----> CP0.config
 21 LOADSEGMENTFIFO CMB0[addr = 1200] ----> CMB0.fifo
 22 LOADPAGESTATE CMB1[addr = 0] ----> CP0.state
 23 LOADPAGEFIFO CMB1[addr = 0] ----> CP0.fifo
 24 SETSEGMENTCONFIGPOINTERS CMB0
 25 CHANGESEGMENTMODE CMB0.mode = SCORE_CMB_SEQSINK
 26 DETECTFULL 1
 27 DETECTDONE 0
 29 STARTSEGMENT 0
 30 STARTPAGE 0
 31 RUNUNTIL

 32 DUMPPAGESTATE CP0.state ----> CMB1[addr = 0]
 33 DUMPPAGEFIFO CP0.fifo ----> CMB1[addr = 0]
 34 LOADPAGECONFIG CMB0[addr = 21680] ----> CP0.config
 35 LOADPAGESTATE CMB0[addr = 21680] ----> CP0.state
 36 LOADPAGEFIFO CMB0[addr = 21680] ----> CP0.fifo
 37 CHANGESEGMENTMODE CMB1.mode = SCORE_CMB_SEQSINK
 38 DETECTFULL 2
 39 CHANGESEGMENTMODE CMB0.mode = SCORE_CMB_SEQSRCSINK
 40 DETECTEMPTY 1
 43 STARTSEGMENT 1
 44 STARTSEGMENT 0
 45 STARTPAGE 0
 46 RUNUNTIL

 47 DUMPSEGMENTFIFO CMB0.fifo ----> CMB0[addr = 1200]
 48 DUMPPAGESTATE CP0.state ----> CMB0[addr = 21680]
 49 DUMPPAGEFIFO CP0.fifo ----> CMB0[addr = 21680]
 50 LOADPAGECONFIG CMB1[addr = 42160] ----> CP0.config
 51 GETSEGMENTPOINTERS CMB[0]
 52 LOADSEGMENTFIFO CMB0[addr = 43360] ----> CMB0.fifo
 53 LOADPAGESTATE CMB1[addr = 42160] ----> CP0.state
```

Figure 3.7: An excerpt from a reconfiguration script used to schedule a four compute page graph onto a small array of 1 CP + 2 CMBs. Three timeslices are shown. `RUNUNTIL` command makes the array run until interrupted by the stall detection.

Runtime Reconfiguration Engine

Figure 3.7 contains an example of a reconfiguration script that the Static Schedule Generator sends to Runtime Reconfiguration Engine. The script contains two types of commands: control commands and reconfiguration commands.

The Runtime Reconfiguration Engine interprets the control commands as actions such as execute until the stall condition is detected, execute for N cycles, jump to a different part of the script, *etc.* The Engine forwards the reconfiguration commands directly to the global array controller. Table 2.1 contains the list of actions handled by the controller. The key actions include transfer configuration bit-stream from a CMB to a CP, set CMB memory bounds registers, and transfer data between off-chip memory and a CMB.

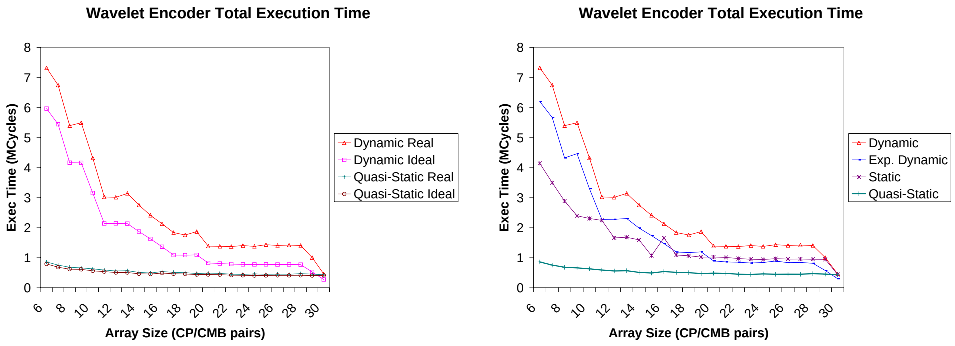
The work evaluates the application execution time and the overheads of scheduling and array reconfiguration by running the Reconfiguration Engine together with a parameterizable cycle-level SCORE array simulator.

3.4.4 Evaluation

This section is a comparative summary of application performance results obtained with fully dynamic versus quasi-static schedulers. The applications chosen for evaluation, JPEG and wavelet codecs, represent a typical workload for the target platform, a FPGA/processor hybrid. Because they combine data-dependent dynamic and static data-flow components, these applications are well suited for performance analysis of the quasi-static scheduler.

Figure 3.8(a) shows the total execution time of the wavelet encoder application for various array sizes, using the dynamic and quasi-static schedulers. Two sets of curves are shown: (1) total application execution time on an ideal array simulation (without scheduling overhead), and (2) on a realistic array simulation (with scheduling overhead). The no-overhead curves demonstrate conclusively that the quasi-static approach yields higher quality schedules than the dynamic approach. The execution time reduction is a factor of 4 on average. In addition, as Figure 3.9 shows, the average timeslice scheduling overhead for the quasi-static scheduler is smaller than dynamic scheduling overhead by a factor ranging from 6 to 12.

Figure 3.8(b) shows the total execution time of the wavelet encoder, with scheduling overhead, for several types of schedulers. The curve marked “Exp. Dynamic” represents an estimate of the application execution time that would result only from reductions in run-time scheduling overhead, and not improvements in scheduling quality. The so-called *static* scheduler shown in the graph is a simple variation on the quasi-static scheduler where stall detection is disabled, so as to use only fixed time-slices. This variation represents point 2 in Figure 3.1, a more static



(a) Quasi-static vs dynamic

(b) Realistic array simulation

Figure 3.8: Wavelet Encoder: Comparison of total application execution time under different schedulers.

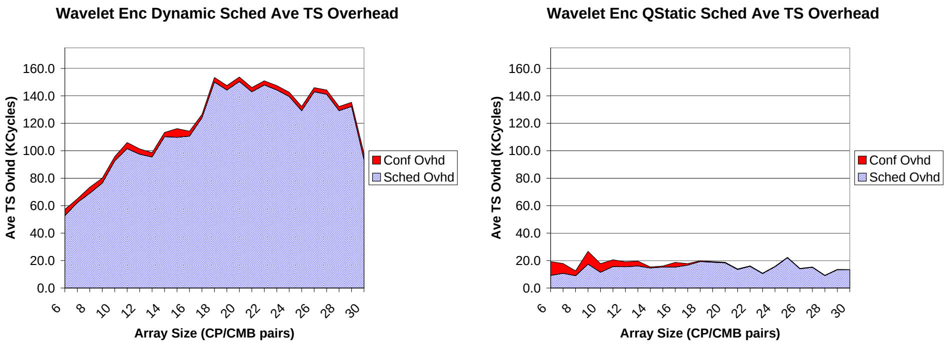
scheduler. Comparing execution times for the quasi-static and static schedulers, we find that the stall-detect feature contributes a factor of about 2 to 3 in improved performance of the quasi-static scheduler.

Interestingly, the fully static scheduler outperforms the dynamic scheduler. Both use the fixed time-slice model, with identical time-slices (250,000 cycles), but the static scheduler gains an edge from its global rather than greedy analysis. Clearly, the perceived advantages of the fully dynamic scheduler, such as the ability to adapt scheduling decisions to match data-flow patterns, are not realized at a feasible scheduling granularity. The timeslice size is constrained by the large scheduling overhead that ranges from 50 to 150 thousand cycles per timeslice.

The superior scheduling quality of the quasi-static scheduler can be attributed to several factors:

- **graph topology:** the quasi-static scheduler has the *global* view of the graph topology, where as the fully-dynamic scheduler is limited to the "frontier" of the breadth-first search.
- **scheduling decisions:** the quasi-static scheduler uses application execution profile to *predict* graph behavior, making an educated guess. The dynamic scheduler adapts its scheduling decisions based on array state.
- **timeslice sizing:** the quasi-static scheduler employs *fine-grained* hardware supported *stall detect* to automatically adapt timeslice size to changes in an application, where as the dynamic scheduler uses a fixed time-slice.

Table 3.1 summarizes total execution times for all four applications evaluated.



(a) Dynamic Scheduler

(b) Quasi-Static Scheduler

Figure 3.9: Wavelet Encoder (30 pages): comparison of average timeslice overhead.

The summary contains the results for the dynamic, the fully static, and the quasi-static schedulers. The complete tables are available in Section B.1 of the Appendix. In comparison to the dynamic implementation, the quasi-static scheduler greatly reduces the fraction of the execution time attributable to scheduling overhead from 30–40% down to 5–10%.

To compare the application execution times obtained with the static schedulers versus the dynamic scheduler, the table contains speedups. Speedups are the reductions in application execution time versus the dynamic scheduler, *i.e.* the speedup for the static scheduler is the ratio of dynamic real and static real execution time. For the quasi-static scheduler it is the ratio of dynamic real and quasi-static real. Notice that speedups are greater on the smaller reconfigurable arrays with limited resources, where improved scheduling quality and efficiency are more critical.

Summary of Total Execution Time (MCycles)

Array Size	Dynamic			Static Real	Quasi-Static			Speedup vs Dyn.	
	Ideal	Real	% Ovhd		Ideal	Real	% Ovhd	Static	Q-Static
Wavelet Encoder (30 pages)									
6	5.968	7.324	*18.5%*	4.144	0.795	0.859	*7.5%*	**1.77**	**8.52**
8	4.166	5.400	*22.9%*	2.887	0.619	0.683	*9.3%*	**1.87**	**7.91**
12	2.142	3.014	*28.9%*	1.662	0.508	0.560	*9.3%*	**1.81**	**5.38**
14	1.872	2.754	*32.0%*	1.595	0.465	0.513	*9.3%*	**1.73**	**5.37**
18	1.086	1.757	*38.2%*	1.067	0.459	0.503	*8.6%*	**1.65**	**3.50**
20	0.827	1.387	*40.4%*	1.025	0.441	0.487	*9.3%*	**1.35**	**2.85**
24	0.776	1.378	*43.7%*	0.942	0.407	0.461	*11.8%*	**1.46**	**2.99**
26	0.774	1.410	*45.1%*	0.956	0.413	0.453	*8.9%*	**1.47**	**3.12**
30	0.275	0.461	*40.2%*	0.442	0.412	0.433	*4.9%*	**1.04**	**1.06**
Wavelet Decoder (30 pages)									
6	9.659	11.631	*17.0%*	3.275	0.812	0.846	*4.0%*	**3.55**	**13.75**
8	7.403	9.137	*19.0%*	2.613	0.721	0.753	*4.3%*	**3.50**	**12.13**
12	6.227	8.187	*23.9%*	1.887	0.608	0.637	*4.5%*	**4.34**	**12.86**
14	5.125	6.887	*25.6%*	1.185	0.530	0.553	*4.1%*	**5.81**	**12.46**
18	1.991	2.803	*29.0%*	1.095	0.511	0.534	*4.3%*	**2.56**	**5.25**
20	2.543	3.683	*30.9%*	1.034	0.497	0.519	*4.4%*	**3.56**	**7.09**
24	1.126	1.717	*34.4%*	1.030	0.499	0.520	*4.1%*	**1.67**	**3.30**
26	0.739	1.167	*36.7%*	1.044	0.517	0.539	*4.0%*	**1.12**	**2.17**
28	0.369	0.536	*31.2%*	0.505	0.479	0.498	*3.7%*	**1.06**	**1.08**
JPEG Encoder (13 pages)									
4	4.832	6.368	*24.1%*	6.756	2.171	2.341	*7.3%*	**0.94**	**2.72**
5	7.134	9.086	*21.5%*	3.460	1.400	1.479	*5.3%*	**2.63**	**6.15**
6	6.458	8.539	*24.4%*	4.682	1.427	1.516	*5.9%*	**1.82**	**5.63**
8	1.655	2.349	*29.6%*	3.173	1.278	1.360	*6.0%*	**0.74**	**1.73**
9	1.672	2.406	*30.5%*	3.216	1.012	1.052	*3.8%*	**0.75**	**2.29**
10	1.635	2.308	*29.2%*	2.179	0.955	0.980	*2.6%*	**1.06**	**2.36**
12	2.621	3.678	*28.7%*	2.194	0.958	0.991	*3.3%*	**1.68**	**3.71**
13	0.797	0.939	*15.1%*	0.899	0.865	0.879	*1.6%*	**1.04**	**1.07**
JPEG Decoder (12 pages)									
3	10.672	13.312	*19.8%*	6.972	2.836	3.004	*5.6%*	**1.91**	**4.43**
4	5.925	8.010	*26.0%*	4.851	1.580	1.701	*7.1%*	**1.65**	**4.71**
5	5.951	7.580	*21.5%*	4.949	1.666	1.796	*7.2%*	**1.53**	**4.22**
7	1.860	2.589	*28.2%*	1.919	1.153	1.192	*3.3%*	**1.35**	**2.17**
8	1.651	2.196	*24.8%*	2.114	0.946	0.979	*3.4%*	**1.04**	**2.24**
9	1.667	2.260	*26.2%*	2.084	0.942	0.975	*3.4%*	**1.08**	**2.32**
11	1.623	2.518	*35.5%*	2.076	0.933	0.965	*3.4%*	**1.21**	**2.61**
12	0.796	0.910	*12.6%*	0.889	0.856	0.869	*1.5%*	**1.02**	**1.05**

Table 3.1: Execution time summary for four applications comparing the scheduler performance. For the dynamic and the quasi-static schedulers, the table shows the percentage of time consumed by the scheduler overhead. For the static and the quasi-static schedulers, the tables also shows the execution time reduction relative to the fully dynamic scheduler.

Chapter 4

Temporal Partitioning

Partitioning is the key component that determines scheduling quality. It is the first step the static schedule generator performs. It divides a virtual page compute graph into schedulable subgraphs to be time-multiplexed on the reconfigurable array.

In SCORE, co-scheduling of neighbor compute nodes is advantageous because it creates compute page pipelines and effectively exploits computation and communication parallelism in hardware. However, an I/O token rate mismatch between neighbors could result in a serious under-utilization of array hardware and therefore increase application execution time. The static schedule generator predicts application behavior by using graph topology and node token emission and consumption rates obtained through profiling. With this information, the scheduler attempts to improve application performance by computing near-optimal graph partitioning that maximizes array hardware utilization.

4.1 Page Activity and Application Performance

Recall the SCORE compute page execution semantics discussed in Section 1.1. A compute page executes only when tokens appear on all inputs required by the current state of its FSM. This is similar to the blocking read in the conventional processor code. On any clock cycle, a compute page either runs or stalls. If the page runs, it consumes the tokens from the required inputs, pushes them through the datapath and emits computed data. If the page stalls, it waits for tokens to arrive at required inputs.

Compute page execution semantics depart from those for SCORE operators because unlike operators pages do not implement non-blocking data token emission (non-blocking write). The compute page can only emit a token on a stream, when the streaming hardware has available space to accept the token. A reconfigurable array has a finite, limited stream token buffering capacity in the interconnect pipeline registers. The consequence of implementing an application on such a device is that a page stalls if the downstream page is a slow token consumer that exerts back-pressure through the hardware streaming interface.

Section 2.1 discussed *bufferlock* as one of the consequences of implementing applications that assume unbounded stream capacity on a device with finite resources. However, *bufferlock* is a rarely occurring condition and is not a problem here because most applications do not assume unbounded stream capacity. The limited network buffering capacity does not present a problem for the scheduler. However, it sets an upper bound on the execution rate of any connected set of co-resident compute pages to be the same as the most active page in the set, the one in the computational critical path.

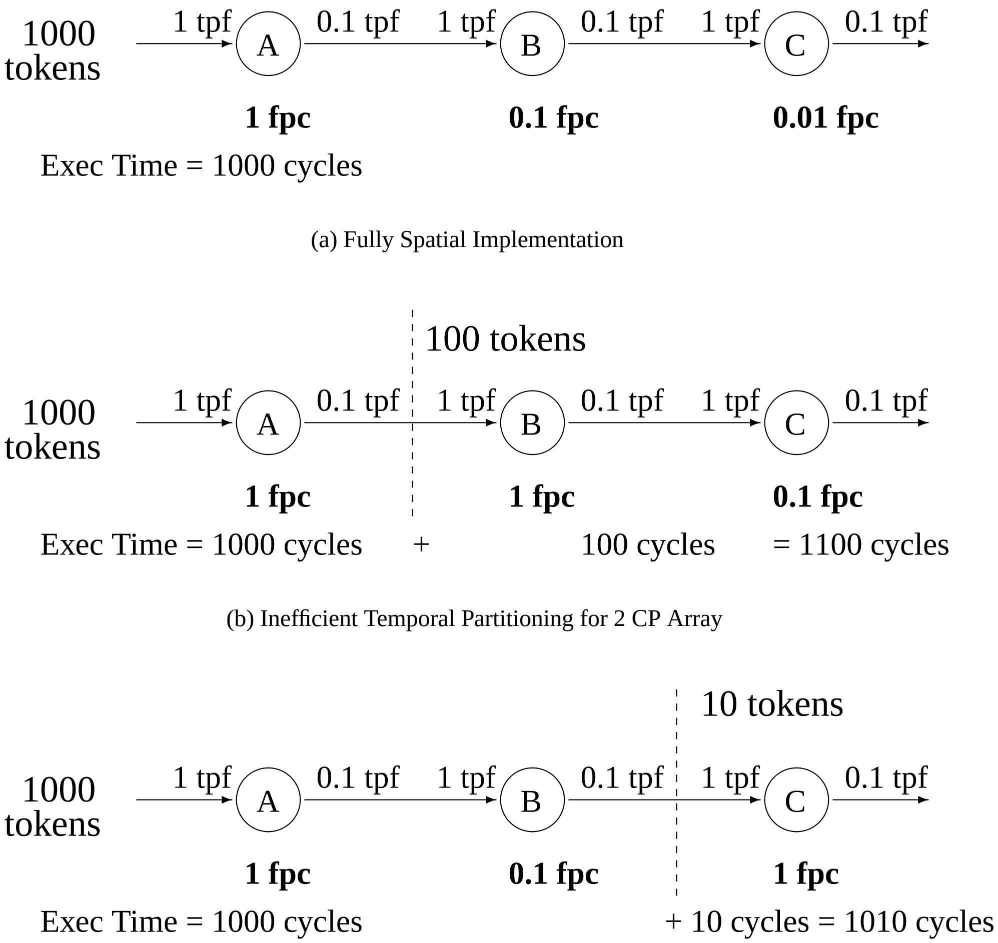
To discuss reconfigurable array resource activity and the total application execution time, let us identify the relationship between these parameters. The following section examines several simple scenarios of temporal graph partitioning and defines the necessary terms. The subsequent section then continues to develop an analytical model that relates array resource activity to the application execution time and guides the temporal partitioning algorithms.

4.1.1 Fundamentals and Metrics

SCORE Graph Temporal Partitioning with the goal to minimize execution time is an NP-hard optimization problem. Note that this work treats temporal partitioning separately from the resource allocation problem, the subject of the next chapter. The goal of temporal partitioning is to minimize the *ideal* application execution time, where ideal indicates a system without scheduling and array reconfiguration overheads. The assumption is that the minimum *ideal* time results in the minimum *realistic* application run time, where all overheads are included. The ramifications of this approach are discussed at the end of this chapter.

Consider a fully spatial implementation of a simple three compute page application as shown in Figure 4.1(a). All pages are resident on the array at the same time. Each page is annotated with an input token consumption rate of 1 token per firing (tpf) and an output token production rate of 0.1 token per firing. This fractional production rate simply implies that a token is produced once per ten page firings because data tokens are atoms. As the previous chapter described, the I/O page rates are obtained by profiling previous application runs.

Figure 4.1(a) shows under each node the expected node firing rates as determined by the balance equations from Synchronous Dataflow (SDF) scheduling [BML96]. *Expected* node firing rate F_n is defined as the number of times per cycle that node n can fire assuming the presence of input data tokens and output



(a) Fully Spatial Implementation

(b) Inefficient Temporal Partitioning for 2 CP Array

(c) Efficient Temporal Partitioning for 2 CP Array

Figure 4.1: The simple example shows token consumption and production rates measured *tokens per firing* (tpf) and expected page firing rates in *firings per cycle* (fpc).

buffering space for the subgraph where node n is resident.

For the graph on Figure 4.1(a), the *expected* node firing rates are 1, 0.1, and 0.01 firings per cycle (fpc) for nodes A, B, and C respectively. The rates illustrate that node A is the most active of three and critical in that it determines the overall array activity in the implementation. This is because no node can fire more than once per cycle. Node A thus determines the application execution time of 1000 cycles. Not surprisingly, 1000 cycles is the same as the number of input tokens, where the pipeline fill time and drain time are ignored for simplicity.

Let us look at the time-multiplexed implementations. Figure 4.1(b) shows one way to partition the three compute page graph for an array with only two physical compute pages. A is the only node in the first timeslice, and its firing rate is the maximum 1 firing per cycle. The nodes in the second timeslice execute independently of node A, and thus their firing rates go up to 1 and 0.1 firings per cycle for B and C respectively. Here, node B is the critical node that determines overall array activity. The total execution time is 1100 clock cycles as shown in the figure.

Figure 4.1(c) shows the other way to partition the same graph. Here, nodes A and B are co-resident while node C runs on its own. This arrangement reduces the total execution time down to 1010 cycles.

4.1.2 Performance Model

This section establishes the relationship between I/O token flow rates, expected page firing rates, and the application performance. Temporal partitioning algorithms use this model to evaluate their results.

Each stream s in a SCORE compute graph $G = (V, E)$ has the token production rate $r_{src}(s)$ and the token consumption rate $r_{snk}(s)$, obtained by profiling previous runs of the application. The token rates range from 0 to 1 tokens per firing (tpf). In a fully spatial implementation, each compute node n has an expected firing rate $F_n \in (0, 1]$, determined by the token flow rates intrinsic to the application. The page firing rates are computed with the balance equation:

$$\left[\forall_{s \in E} . F_{src(s)} r_{src}(s) = F_{snk(s)} r_{snk}(s)\right] \wedge \left[\forall_{n \in V} . F_n \leq 1\right] \tag{4.1}$$

Expected node firing rate F_n is the upper bound on the node's ability to fire in a graph, constrained by the application algorithm that dictates the token flow rates.

Although, a node firing rate on its own does not determine the application execution time, a metric such as the average firing rate of all array compute pages does. Recall from Table 2.2 that $ts(i)$ represents the set of compute nodes and streams in the timeslice i. A node's firing rate in the timeslice i is bound by the most active node in that partition, *i.e.* the expected firing rate of the node n is normalized to the most active node:

$$F_n^{ts(i)} = \frac{F_n}{\max_{m \in ts(i)} \{F_m\}}, \text{if } n \in ts(i) \tag{4.2}$$

As was mentioned previously, this rate assumes the presence of input tokens to the subgraph scheduled in timeslice i and the availability of the buffer space to emit results. Neither is an issue. The input tokens to be processed come from *stitch* buffers that connect timeslice i to the timeslice $i - 1$. The buffer space to store intermediate computation results is available in the *stitch* buffers that connect timeslices i with $i + 1$.

The array activity during timeslice i is the average CP firing rate. It can be computed with

$$A_{ts(i)} = \frac{1}{P} \times \sum_{n \in ts(i)} \frac{F_n}{\max_{m \in ts(i)} \{F_m\}} \tag{4.3}$$

which can be rearranged as

$$A_{ts(i)} = \frac{1}{P} \times \frac{1}{\max_{m \in ts(i)} \{F_m\}} \times \sum_{n \in ts(i)} F_n \tag{4.4}$$

Computing the array activity during the entire application execution involves more than averaging the activities of individual timeslices. The timeslice length varies, and therefore the total array activity is a weighted sum of timeslice activities. To compute timeslice length, recall from Section 2.3.1 that to efficiently utilize CMB memory the buffer sizes B_i should be directly proportional to the rates p_i of the streams they buffer (Equation 2.8):

$$\forall_{i \neq j} \left(\frac{p_i}{B_i} = \frac{p_j}{B_j} \right) \tag{4.5}$$

If stream $s \in ts(i)$, the length of the timeslice i can be defined in terms of the computed stream buffer size B_j:

$$T_{ts(i)} = \frac{B_s}{r_{snk}(s) \times \frac{F_{snk(s)}}{\max_{m \in ts(i)} \{F_m\}}} \tag{4.6}$$

which specifies that the length of the timeslice is the number of cycles required to consume all B_s tokens from the stream. The denominator of the fraction is the rate of token consumption adjusted by the expected firing rate of the node $snk(s)$.

Equation 4.6 serves as the base to redefine T_{ideal}, first defined in Equation 2.6:

$$\begin{aligned} T_{ideal} &= \sum_{i \in [1,S]} T_{ts(i)} && (4.7) \\ &= \sum_{i \in [1,S]} \frac{B_s}{r_{snk}(s) \times \frac{F_{snk(s)}}{\max_{m \in ts(i)}\{F_m\}}} && (4.8) \end{aligned}$$

With the help of $T_{ts(i)}$ and T_{ideal}, the array activity during the entire application execution is defined as a weighted sum of CP activities for each timeslice:

$$\begin{aligned} A &= \sum_{i \in [1,S]} \frac{T_{ts(i)}}{T_{ideal}} \times A_{ts(i)} \\ &= \frac{1}{T_{ideal}} \times \sum_{i \in [1,S]} \frac{B_s}{r_{snk}(s) \times \frac{F_{snk(s)}}{\max_{m \in ts(i)}\{F_m\}}} \times \frac{\sum_{n \in ts(i)} F_n}{P \max_{m \in ts(i)} \{F_m\}} \\ &= \frac{1}{T_{ideal}} \times \sum_{i \in [1,S]} \left(\frac{B_s}{r_{snk}(s) \times F_{snk(s)}} \times \frac{\sum_{n \in ts(i)} F_n}{P} \right) && (4.9) \end{aligned}$$

The last equation demonstrates the intuitive, inverse relationship between the average array activity A and the application execution time T_{ideal}.

This model evaluates the temporal partitioning quality of the algorithms in this work. To compute the expected array activity, the static schedule generator computes the relative buffer sizes, B_i, using the Equation 4.5. Then T_{ideal} is computed using Equation 4.8, and finally, the array activity A is calculated with the model in Equation 4.9.

Let us revisit the examples in Figures 4.1(b) and 4.1(c) to compute average array activity A for both scenarios.

$$\begin{aligned} A_1 &= \frac{1}{1100} \times \left(1000 \times \frac{1}{2} + 100 \times \frac{1+0.1}{2} \right) = 0.504 && (4.10) \\ A_2 &= \frac{1}{1010} \times \left(1000 \times \frac{1+0.1}{2} + 10 \times \frac{1}{2} \right) = 0.549 && (4.11) \\ & && (4.12) \end{aligned}$$

As expected, the second temporal partitioning results in the higher array CP utilization and produces shorter application execution time.

4.1.3 Measuring Array Activity

The model in the previous section estimates the average array activity by the scheduling algorithms in the system. However, the activity can also be measured by the

SCORE array simulator to evaluate scheduling quality for a given device size. The array simulator records the number of cycles each compute page (CP) fired during execution, and computes an *observed* average CP utilization $\overline{U}$:

$$\overline{U} = \frac{\sum_{i=1}^{P} U_i}{T \times P} \tag{4.13}$$

where P is the the number of CPs on the array, U_i is the number of cycles that CP i fired, and T is the total application execution time.

4.2 Algorithms for Temporal Partitioning

While total application execution time depends on the size of the input dataset, average CP utilization depends only on the schedule. Co-resident pages and the token flow rate mismatches between them affect the schedule quality. The *Partitioner* algorithms use the model in Equation 4.9 to estimate average CP utilization for a given candidate partitioning set and attempt to maximize utilization. The goal is not to achieve 100% CP utilization but to attain the highest possible CP utilization for a specific array size. This results in the lowest total application execution time.

A temporal partitioning algorithm should also avoid breaking any cycles in the dataflow graph. The exact behavior of an application is not known to the scheduler, and thus it must assume that there exists a close dependency between compute pages and memory segments that form a dataflow cycle. Therefore, if the cycle is split between temporal partitions, every communication between the nodes in different timeslices involves a costly reconfigurable array context switch.

Optimal graph partitioning under multiple independent simultaneous constraints such as CP/CMB count is an NP-hard optimization problem. To understand the problem in detail, this work develops two heuristic partitioners, which are compared against an optimal exhaustive search partitioner. Before we delve into details of the heuristics, let us recall the way to evaluate hardware resource requirements for a temporal partition. A temporal partition requires: one CP per virtual compute page, one CMB per user-defined memory segment, and one CMB per each stream that cross the partition boundary to hold the *stitch* buffer with intermediate computation results.

4.2.1 Topological Partitioner

Topological partitioner uses a simple greedy packing algorithm with a precedence constrained graph traversal order. Essentially, the algorithm starts by topologically sorting graph nodes, then iterates over the resultant node list forming schedulable

1. Graphs
   - Graph of compute nodes (pages and segments): $G_n = (V_n, E_n)$
   - Graph of compute node clusters: $G_c = (V_c, E_c)$, where $c \in V_c$ is a cluster, or a set of compute graph nodes. All clusters are nonempty $|c| \geq 1$.

2. Temporal Partition Tables
   - An array of node sets, s.t. $\mathcal{P}_n[k]$ is the set of nodes in temporal partition k.
   - An array of cluster sets, s.t. $\mathcal{P}_c[k]$ is the set of clusters in temporal partition k.

3. Resource Allocation Map
   - CP Resource Map $CP[ts][i]$ represents the contents of CP_i in the timeslice ts. The entry can contain either a compute page or $\emptyset$, if it is available for allocation. This entry has an attribute $CP[ts][i].locked \in [\textbf{true}, \textbf{false}]$, which indicates whether this entry can be evicted or not.
   - CMB Resource Map $CMB[ts][i][start..stop]$ represents the contents of the address range $(start..stop)$ of CMB_i in the timeslice ts. Possible operations include checking if the range exists and intersecting one address range with another. $CMB[ts][i][].locked$ indicates whether this entry can be evicted by the resource allocation algorithm. Another attribute is $CMB[ts][i][].entry\_type = \{segment, config\}$. $segment$ indicates that the CMB memory block contains a user-specified segment or a *stitch* segment. $config$ indicates that the block contains a configuration bitstream and a context for a virtual compute page.

Figure 4.2: Key data structures in the scheduler algorithms

subgraphs. Possessing only a very limited global view of the entire graph, this algorithm improves the partitioning quality by relying on two special clustering passes: cycle clustering and minimize IO clustering.

Cycle Clustering. The first pass identifies the dataflow cycles in the compute graph. It clusters the nodes that belong to the same cycle together, turning the original compute graph into the directed acyclic graph (DAG) of node clusters. The following pseudo-code demonstrates the basic operation of the cycle clustering. The function COMPUTE_CYCLES uses depth-first search to find cycles in the compute graph [CLR90]. It returns a list of cycles found, which is used to construct the graph of clusters $G_c = (V_c, E_c)$. The algorithm creates simple single-node clusters for those compute nodes that do not belong to a cycle.

```
Function CLUSTER_CYCLES ($G_n = (V_n, E_n)$)

# Compute the list of loops in the $G_n$
# Each cycle in CycleList is a set of nodes
CycleList $\Leftarrow$ COMPUTE_CYCLES($G_n$)

for all n $\in$ NodeGraph do  # mark all nodes as unclustered
   Clustered[n] $\Leftarrow$ false
end for

$G_c = (V_c, E_c) \Leftarrow (\emptyset, \emptyset)$
for all cycle $\in$ CycleList do
   cluster $\Leftarrow \emptyset$
   for all n $\in$ cycle do
      Clustered[n] $\Leftarrow$ true
      cluster.ADDNODE(n)
   end for
   # add cluster to the cluster graph data structure
   # establish edges to neighbor clusters
   $G_c$.ADDCLUSTER(cluster)
end for

for all n $\in$ NodeGraph do  # all nodes that are not in a cycle get own cluster
   if Clustered[n] = false then
      cluster $\Leftarrow \emptyset$
      cluster.ADDNODE(n)
      $G_c$.ADDCLUSTER(cluster)
```

```
  end if
end for

return $G_c$
```

Minimize I/O clustering. The second pass strives to decrease the number of streams crossing temporal partition boundary. Because this partitioner traverses the graph in topological order, its view of the entire graph is limited. This clustering pass merges two neighbor clusters if the resulting cluster I/O stream count is lower than the aggregate I/O stream count of the two separately. This helps eliminate undesirable, high bandwidth cuts in the compute graph.

The algorithm takes the cluster graph $G_c = (V_c, E_c)$ obtained from the cycle clustering, physical compute page count P, and CMB count Y. It iterates through all cluster nodes, merging together those neighbors where a cluster join would reduce I/O stream count. This process is repeated until no more joins are possible.

```
Function CLUSTER_MIN_IO ($G_c = (V_c, E_c)$, $P$, $Y$)

# repeat join passes until no more nodes can be joined
repeat
  # attempt to find clusters that can be joined w/o exceeding array size
  $JoinCount \Leftarrow 0$
  for all $n \in V_c$ do  # iterate through all clusters
    for all $(n, l) \in E_c$ do  # iterate through all downstream neighbors of $n$
      # compute the number of IOs in cluster $n$
      $NeighborSet \Leftarrow \{m|(n, m) \in E_c\} \cup \{m|(m, n) \in E_c\}$
      $OldIOCount \Leftarrow |NeighborSet|$
      # compute resource and IO count, if $n$ and $l$ join
      $(cps, cmbs, NewIOCount) \Leftarrow$ CHECK_JOIN($n, l$)
      if $cps \leq P \wedge cmbs \leq Y \wedge NewIOCount < OldIOCount$ then
        $G_c$.JOIN_CLUSTERS($n, l$)  # join cluster $l$ to $n$
        $G_c$.DELETE($l$)
        $JoinCount \Leftarrow JoinCount + 1$
      end if
    end for
  end for
until $JoinCount = 0$
return $G_c$
```

The partitioner computes the array $\mathcal{P}_c[]$ using the cluster graph obtained from the clustering passes above. The array contains all members of the temporal parti-

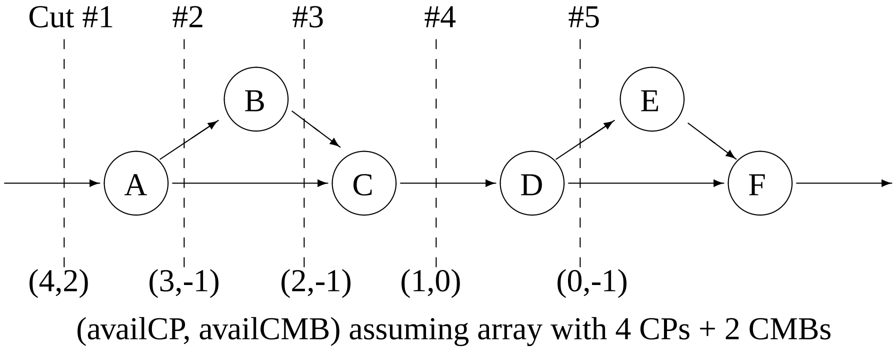


Figure 4.3: Example to illustrate the need for the two step topological partitioning. To make the first partition, the topological algorithm attempts cuts 1–5, but rolls back to cut 4 to satisfy CMB count constraint.

tions, such that $\mathcal{P}_c[k]$ is the set of clusters in the temporal partition k.

Recall that the cycle clustering pass yields an acyclic cluster graph. The Topological Partitioning algorithm below computes the topological sort ($TraversalList$) of clusters, which determines the order the partitioner examines the clusters. A partition expands as the algorithm adds clusters one at a time from the $TraversalList$. Because this process is constrained only by the CP count (lines 11–18), the algorithm attempts to pack as many clusters in a partition as possible, improving the expected array activity and utilization. However, since the pseudo-code lines 11–18 compute the partition without considering CMB count, the partition may not fit on the array. The subsequent part of the algorithm (lines 20–29) removes the clusters from the partition one at a time in the reverse topological order until both CP and CMB constraints are satisfied. This guarantees that the resulting partitions meet the precedence constraints. The algorithm repeats these two operations until all graph nodes belong to a partition.

The need for this two step process can be illustrated with an example in Figure 4.3. Assume that the array contains 4 physical compute pages (CPs) and 2 configurable memory blocks (CMBs). The $TraversalList$ forces the compute nodes to be considered in the order $\{A, B, C, D, E, F\}$. A new cut is formed after a node is removed from the $TraversalList$. The algorithm first makes cuts 1 through 5, filling all four array CPs (lines 11–18). Then it rolls the partition back to 3 pages (A, B, C) to satisfy the CMB count constraint (lines 20–29). The second temporal partition expands starting from node D and includes the remaining three nodes in the graph.

Unless the first step in the algorithm ignores CMB count, topological partitioning either leads to inefficient results by under-utilizing array CPs, or as the case is in this example with only 2 CMBs, simply fails.

```
Function PERFORM_TOPO_PARTITION ($G_c = (V_c, E_c)$, $P$, $Y$)

# Compute list of clusters sorted in topological order
$TraversalList \Leftarrow$ TOPO_SORT($G_c$)
$currTimeslice \Leftarrow 0$
while SIZE($TraversalList$) $> 0$ do
    $availCP \Leftarrow totalCP$ # keep track of available resources
    $availCMB \Leftarrow totalCMB$
    $schedClusterList \Leftarrow \emptyset$ # list of clusters that may be scheduled

    # first, schedule all possible pages even if availCMB becomes less than 0
    # this makes sure that compute pages are packed as much as possible
    while $availCP > 0 \wedge SIZE(TraversalList) > 0$ do
        $cluster \Leftarrow POP(TraversalList)$
        $PUSH(schedClusterList, cluster)$
        # available resources if all $schedClusterList$ clusters are co-resident
        $(availCP, availCMB) \Leftarrow GET\_AVAIL\_RES(schedClusterList, P, Y)$
    end while

    # The clusters in $schedClusterList$ do not exceed the CP count,
    # but may not be schedulable because they may require more CMBs
    # than available. Remember that a CMB is required for each stream
    # crossing the timeslice boundary. Unschedule from the back of
    #  the $schedClusterList$ until the set is schedulable.
    while $availCP < 0 \vee availCMB < 0$ do
        $cluster \Leftarrow POP(schedClusterList)$
        $PUSH(TraversalList, cluster)$
        $(availCP, availCMB) \Leftarrow GET\_AVAIL\_RES(schedClusterList, P, Y)$
    end while

    $\mathcal{P}_c[currTimeslice] \Leftarrow nodeList$
    $currTimeslice \Leftarrow currTimeslice + 1$
end while
return $\mathcal{P}_c$
```

Algorithm Complexity Analysis. The function CLUSTER_CYCLES performs a depth-first search of the compute graph to identify all its cycles. Then it enumerates all nodes while clustering them. The complexity of the algorithm is linear $O(|E_n| + |V_n|)$.

The loops on lines 7 and 8 of the CLUSTER_MIN_IO algorithm make $|E_c|$ total repetitions because every edge is visited once. There must be at least one join made for the main loop on lines 7–20 to repeat. Since every cluster join reduces the total number of clusters by 1, there can at most be $|V_c|$ iterations of the **repeat – until** loop. This results in the worst case algorithm complexity of $O(|V_c| \times |E_c|)$.

The topological sort in PERFORM_TOPO_PARTITION is implemented with a depth-first search, which runs in time proportional to $O(|E_c|)$. The remainder of the algorithm iterates over the clusters in the $TraversalList$. In the worst case, all but the first cluster scheduled in a partition is rolled back in lines 20–29 ($|V_c|^2$). Hence the topological partitioner in the worst case has the complexity of $O(|E_c| + |V_c|^2)$.

4.2.2 Balanced N-way Mincut

This flow-based mincut partitioning algorithm is based on Wong's temporal partitioning for FPGA circuits [LW98]. Unlike the FPGA circuits, SCORE compute graphs contain clear precedence constraints. The algorithm below has been adapted to enforce them. This work considers compute graphs of pages and segments, where each edge has a capacity of 1. This capacity represents 1 CMB that is required to buffer the content of the stream that crosses the timeslice boundary. Other weight assignments are also possible and will be discussed in Section 4.4.

In its core, the algorithm performs a minimum capacity cut on the graph and examines resulting partitions. The algorithm grows or shrinks the partition with the nodes of the highest scheduling precedence until the partition fits on the array. To grow or shrink a partition, the algorithm augments or evicts a selected node and repeats the mincut. The partitioner uses the average CP utilization model (Equation 4.9) to select nodes to move between partitions. Its goal is to maximize predicted average CP utilization by selecting the candidate to augment or evict that will improve array utilization for the duration of the timeslice. Once the algorithm completes a partition, it repeats the partitioning process for the part of the compute graph that has not yet been scheduled.

Let us look at several specific details of PERFORM_MINCUT_PARTITION. This flow-based mincut algorithm is based on the mincut-maxflow theorem that states that the maximal amount of a flow is equal to the capacity of a minimal cut. The algorithm operates by repeatedly augmenting flows from a source to a target node in the network reaching the maximum flow. The mincut contains a subset of the capacity saturated edges. One peculiarity about the flow-based mincut is the direction of the flow. The mincut computed by the flow-based algorithm consists of edges that are saturated with the flow from the source to the target node. The oppositely directed edges do not contribute to the mincut capacity, and this algorithm takes advantage of this peculiarity.

With the help of special infinite capacity edges, one can invalidate certain cuts for the flow-based mincut algorithm. Consider the lines 8–10 in the algorithm below. For every edge (n, m) in the original compute graph, the algorithm adds a special reverse infinite capacity edge (m, n). Since the (n, m) sets the precedence constraint between its source and sink, a cut along (n, m) is valid since it schedules n in the current timeslice and m in the subsequent. However, the opposite is not true. A cut which puts m in the current timeslice and n in the subsequent is not valid. The infinite capacity edge never saturates and thus does not allow the MINCUT procedure to emit a cut that violates the precedence constraints.

Function PERFORM_MINCUT_PARTITION begins with a topological sort of the cluster graph. The topologically sorted list of clusters allows the algorithm to quickly select the source and the target cluster for the flow-based mincut partitioner. The code in lines 13–22 drives the partitioner. On every iteration of the loop, the function COMPUTE_PARTITION emits a set of nodes to be scheduled in the timeslice ts and the updated $TopoList$, which contains the clusters still waiting to be scheduled. The loop terminates when the contents of the $TopoList$ fit on the array.

Function COMPUTE_PARTITION plays the key role in this partitioner. It begins with a mincut that splits the clusters in the graph into two sets. The set on the side of the src cluster ($srcClustList$) is considered for the partition because it has higher precedence than the nodes in the $trgt$ side. The algorithm evaluates the physical resource requirements of the clusters in $srcClustList$. If the partition is too small, *i.e.* CPs are available, the algorithm augments the partition: (1) all clusters in $srcClustList$ are merged to the src with infinite capacity edges (lines 54–59); (2) a candidate is selected from the nodes on the $trgt$ side of the partition (lines 60–75) to provide the highest expected array activity for the partition; (3) the candidate is merged with the src node via infinite capacity edge (lines 80–86); (4) the mincut partition is repeated.

If the partition is too large, the opposite occurs: the algorithm selects and evicts a node from $srcClustList$ that does not violate precedence constraints and leads to the highest possible array activity (lines 91–117). Each iteration of the main algorithm loop merges the chosen clusters via infinite capacity edges to either src or $trgt$ clusters. The size of the cut returned by the MINCUT function on line 45 increases with each iteration. Therefore, it is possible that a cut exceeds CMB count. In that case (lines 46–51), the algorithm rolls back to the previous best solution stored in $PrevPart$.

```
Function PERFORM_MINCUT_PARTITION ($G_c = (V_c, E_c)$, $P$, $Y$)

# Compute list of clusters sorted in topological order
$TopoList \Leftarrow$ TOPO_SORT($G_c$)
for all $e \in E_c$ do
    # Annotate edges with 1 to represent 1 CMB/stream buff requirement
    $w[e] \Leftarrow 1$
    # $\infty$ cap reverse edges prevent mincuts that violate precedence
    $NewEdge \Leftarrow G_c$.ADD_EDGE($snk(e), src(e)$)
    $w[NewEdge] \Leftarrow \infty$
end for

$ts \Leftarrow 0$
$(availCP, availCMB) \Leftarrow$ GET_AVAIL_RES$(TopoList, P, Y)$
# function COMPUTE_PARTITION removes the clusters from
# the beginning the $TopoList$. After some number of iterations,
# it the contents of the $TopoList$ will fit on array
while $availCP < 0 \vee availCMB < 0$ do
    $(\mathcal{P}_c[ts], TopoList) \Leftarrow$ COMPUTE_PARTITION($G_c, TopoList, w[], P, Y$)
    $(availCP, availCMB) \Leftarrow$ GET_AVAIL_RES$(TopoList, P, Y)$
    $ts \Leftarrow ts + 1$
end while
if $SIZE(TopoList) > 0$ then  # the last partition
    $\mathcal{P}_c[ts] \Leftarrow TopoList$
end if
return $\mathcal{P}_c$


Function COMPUTE_PARTITION($G_c, TopoList, w[], P, Y$)

# a list of "special" edges, which are temporarily added to the graph
# and will be removed when the routine exits
$specialEdgeList \Leftarrow \emptyset$
# first and last elmts are the source and target node for the flow based mincut
$src \Leftarrow$ FIRST($TopoList$)
$trgt \Leftarrow$ LAST($TopoList$)

# In this iterative algorithm, the cuts grow larger in b/w with each iteration.
# $PrevPart$ contains the best partition from the previous iteration.
$PrevPart \Leftarrow \emptyset$
```

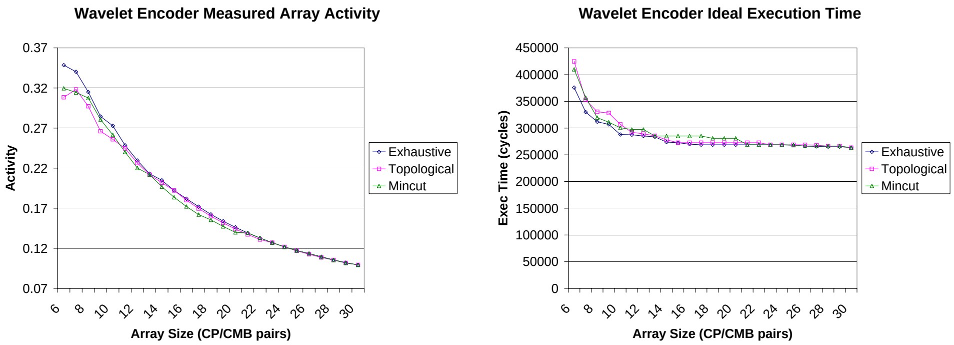
```
loop
  # The flow-based mincut algorithm partitions $G_c$, such that $src$ and $trgt$
  # clusters are in separate partitions. It returns two distinct sets of clusters,
  # one with the nodes on the $src$ side and the other on the $trgt$ side.
  $(srcClustList, trgtClustList) \Leftarrow$ MINCUT ($G_c, src, trgt, w$)
  # compute available resource if all members of $srcClustList$ are scheduled
  $(availCP, availCMB) \Leftarrow$ GET_AVAIL_RES$(srcClustList, P, Y)$
  if $availCMB < 0$ then  # exceeded CMB reqs (cut is too large)
    # revert to the previously computed partition, stored in $PrevPart$
    break # quit the loop
  end if

  if $availCP > 0$ then  # try adding more clusters to the partition
    for all $n \in srcClustList$ do
      # merge $n$ to $src$ by adding special edge of $\infty$ capacity
      $specialEdge \Leftarrow G_c$.ADD_EDGE($src, n$)
      $w[specialEdge] \Leftarrow \infty$
      $PUSH(specialEdgeList, specialEdge)$
    end for
    # make a list of valid candidates to augment to the partition
    # a valid candidate $\Rightarrow$ all predecessors have been scheduled
    $candList \Leftarrow$ AUGMENT_CANDS($trgtClustList, TopoList, P, Y$)
    if SIZE($candList$) $> 0$ then
      $ActivVec \Leftarrow \emptyset$
      for all $c \in candList$ do
        # Compute timeslice activity if $c$ is scheduled with $srcClustList$
        $ActivVec[c] \Leftarrow$ ARRAY_ACTIV($srcClustList \cup \{c\}, P, Y$)
      end for
    end if
    # it is possible that the original (prior to augmentation) partitioning
    # is a better choice than the new cuts
    $OrigArrayActiv \Leftarrow$ ARRAY_ACTIV($srcClustList, P, Y$)
    $ActiveVec[0] \Leftarrow OrigArrayActiv$
    # select a candidate with the highest expected array activity
    SELECT $cand$, s.t. $ActiveVec[cand] = max_i\{ActiveVec[i]\}$

    # save the current partition in case the next iteration fails
    $PrevPart \Leftarrow srcClustList$

    # Temporarily merge the $cand$ to $src$, so that the mincut
    # performed in the next iteration puts $src$ and $cand$ in the same partition
    **if** $cand \neq 0$ **then**
      $specialEdge \Leftarrow G_c$.ADD_EDGE($src, cand$)
      $w[specialEdge] \Leftarrow \infty$
      $PUSH(specialEdgeList, specialEdge)$
    **else**
      **break** # partition is good, $OrigArrayActiv$ was chosen
    **end if**
  **else if** $availCP = 0$ **then** # bulls eye
    $PrevPart \Leftarrow srcClustList$
  **else if** $availCP < 0$ **then** # CP reqs are too high, evict a node
    **for all** $n \in trgtClustList$ **do**
      # Merge $n$ to the $trgt$ cluster to prevent Mincut algorithm from
      # separating them in the next iteration.
      $specialEdge \Leftarrow G_c$.ADD_EDGE($n, trgt$)
      $w[specialEdge] \Leftarrow \infty$
      $PUSH(specialEdgeList, specialEdge)$
    **end for**

    $candList \Leftarrow$ EVICT_CANDS($srcClustList, TopoList, P, Y$)
    **if** SIZE($candList$) $> 0$ **then**
      $ActivVec \Leftarrow \emptyset$
      **for all** $c \in candList$ **do**
        $ActivVec[c] \Leftarrow$ ARRAY_ACTIV($srcClustList - \{c\}, P, Y$)
      **end for**
      # select a candidate with the highest expected array activity
      SELECT $cand$, s.t. $ActiveVec[cand] = max_i\{ActiveVec[i]\}$

      # Merge the $cand$ to the $trgt$ node to guarantee that is is evicted
      # from $srcClustList$ in the next iteration.
      $specialEdge \Leftarrow G_c$.ADD_EDGE($cand, trgt$)
      $w[specialEdge] \Leftarrow \infty$
      $PUSH(specialEdgeList, specialEdge)$
    **else** # no $cand$ can be evicted, go to the previous successful iter.
      **break**
    **end if**
  **end if**
**end loop**

# break in the loop goes here. $PrevPart$ contains the last successful
# partition attempt. Now, remove all nodes in the partition from the $TopoList$.
for all $n \in PrevPart$ do
    REMOVE($TopoList, n$)
end for

# Remove all special $\infty$ edges that were used to merge nodes together,
# because they could interfere with mincut partitioning in
# the subsequent invocations of this routine.
for all $e \in specialEdgeList$ do
    $G_c$.DELETE_EDGE($e$)
end for
# return the partition and the updated topologically sorted list of clusters
return ($PrevPart, TopoList$)
```

PERFORM_MINCUT_PARTITION is based on Wong's mincut multi-way partitioning described in [LW98] in that it repeatedly performs a mincut operation. Although each MINCUT algorithm typically requires constructing a complete residual flow network, Wong avoids this by sharing the residual flow network between MINCUT invocations. The residual flow network is thus constructed incrementally. This reduces the running time of a single invocation of the COMPUTE_PARTITION function to only $O(|V_c| \times |E_c|)$. Thus the running time of the entire mincut-based partitioner is $O(S \times |V_c| \times |E_c|)$, where S is the number of temporal partitions and $S \geq \lceil \frac{N}{P} \rceil$.

4.2.3 Exhaustive Search

To obtain a reference for partitioning quality, the system developed in this work includes an algorithm that examines every valid graph partitioning to find the one with highest average CP utilization. This algorithm helps evaluate the quality of the two heuristic partitioners described previously.

In general, the number of candidate schedules the partitioner must examine grows exponentially with graph size. However, the complexity of the exhaustive search is largely bounded by graph precedence constraints and simple branch-and-bound heuristics that avoid clearly inefficient solutions. Nevertheless, computing optimal partitioning for a specific array size may take hours. For example, for the 30 page wavelet encoder mapped on 6 CP array, the algorithm examines in excess of 101 million candidates and consumes more than 18 hours on a P3 500Mhz system.



(a) Average Array Activity

(b) Execution Time

Figure 4.4: Wavelet Encoder (30 pages) performance summary for temporal partitioning algorithms.

4.3 Evaluation

This section examines the quality of results for topological and mincut-based partitioners. The total application execution time is shown for the ideal reconfigurable array without reconfiguration or scheduling overhead.

Critical to the quasi-static methodology is the off-line schedule generator's ability to accurately predict average array activity. The experiments in this work show that the static schedule generator predicts the *measured* average CP utilization with less than 5% error for all applications studied. The results discussed below were obtained through measurements on the SCORE reconfigurable array simulator.

Not surprisingly, the plot in Figure 4.4(a) demonstrates that, of all the partitioners, the exhaustive search produces the schedule with highest average CP utilization. A surprising result is that the topological and mincut-based heuristic partitioners in *the worst case* perform within 17% of the optimal in terms of measured average array activity across all applications. Another unexpected result is that neither heuristic partitioner consistently outperforms the other, although they differ greatly in algorithmic complexity.

Figure 4.4(b) shows wavelet encoder total execution time using the different partitioners. The expected inverse relationship between the total execution time and the average CP utilization holds, validating the hypothesis that increased utilization is a good predictor of reduced execution time.

Table 4.1 summarizes the comparison between temporal partitioning algorithms. Refer to the appendix section B.2 for the complete comparison. Table 4.1 shows the ratio between the execution times for the topological and exhaustive search partitioners and the ratio between the mincut-based and exhaustive partitioners. In the worst case, both heuristics have a performance penalty of 60%, while typically offer application performance within 10% of the ideal.

Recall that the previous chapter reveals that improvements in performance between dynamic and static schedulers exceed what one would expect purely from reductions in the scheduler run-time overhead. The performance results show that the shift to the static scheduling methodology yields improvements in schedule quality.

These improvements are attributable to the global view that the static schedule generator has of the compute graph. Rather than focusing on token availability as the dynamic implementation does, the static scheduler predicts token flow from application execution profiles. Section 4.1 emphasizes the importance of temporal partitioning that attempts to improve array activity. The static scheduler generally makes more accurate decisions about the way to temporally partition the graph.

4.4 Open Issues

The two heuristics employed in the *Partition* module of the quasi-static scheduler are by no means all that can be used for effective temporal partitioning. Although the results in this chapter show that the heuristics perform well on average, it is possible to study several other options.

Consider the balanced N-way mincut heuristic. The implementation in this work gave all graph edges the capacity of 1 to perform the min-cut that minimizes the number of CMBs required to buffer the intermediate computation state. One can consider using other metrics, such as relative stream buffer sizes to minimize the total capacity of the required *stitch* buffers. This could potentially improve resource utilization and help the resource allocation algorithms, the subject of the following chapter.

This work treats the temporal partitioning problem separately from the resource allocation problem. Although to obtain the optimal solution to the entire scheduling problem stated in Section 2.3.2, both partitioning and resource allocation must be solved simultaneously. The reason to separate these is simply to keep the problems tractable. The ramifications of this approach are unfortunately difficult to evaluate due to a very large solution space.

Summary of Temporal Partitioning Algorithms

Array Size	Execution Time			Improvement		Measured Array Activity		
	Exhaustive	Topological	Mincut	Topo/Exh	Min/Exh	Exhaustive	Topological	Mincut
Wavelet Encoder (30 Pages)								
6	375808	424704	409856	**1.13**	**1.09**	0.35	0.31	0.32
8	311808	330496	319488	**1.06**	**1.02**	0.32	0.30	0.31
10	288000	306944	300544	**1.07**	**1.04**	0.27	0.26	0.26
12	285184	289280	297472	**1.01**	**1.04**	0.23	0.23	0.22
14	273746	278016	285184	**1.02**	**1.04**	0.20	0.20	0.20
16	269881	272640	285184	**1.01**	**1.06**	0.18	0.18	0.17
18	268800	272640	280832	**1.01**	**1.04**	0.16	0.16	0.16
20	268800	272640	280832	**1.01**	**1.04**	0.15	0.14	0.14
22	268800	272640	268800	**1.01**	**1.00**	0.13	0.13	0.13
Wavelet Decoder (27 Pages)								
6	315217	373094	333201	**1.18**	**1.06**	0.38	0.32	0.36
8	295221	391354	323966	**1.33**	**1.10**	0.31	0.23	0.28
10	287008	323136	306231	**1.13**	**1.07**	0.25	0.22	0.24
12	286004	332647	306256	**1.16**	**1.07**	0.21	0.18	0.20
14	272162	435358	306277	**1.60**	**1.13**	0.19	0.12	0.17
16	269600	298824	306293	**1.11**	**1.14**	0.17	0.15	0.15
18	269592	313954	339066	**1.16**	**1.26**	0.15	0.13	0.12
20	269600	282684	272160	**1.05**	**1.01**	0.14	0.13	0.13
22	272163	296774	272158	**1.09**	**1.00**	0.12	0.11	0.12
JPEG Encoder (13 Pages)								
4	1394432	1611008	1394176	**1.16**	**1.00**	0.45	0.39	0.45
5	1107712	1624320	1394176	**1.47**	**1.26**	0.46	0.31	0.36
6	1116645	1116672	1116672	**1.00**	**1.00**	0.38	0.38	0.38
7	1065366	1096704	1065434	**1.03**	**1.00**	0.34	0.33	0.34
8	1065364	1072896	1065412	**1.01**	**1.00**	0.30	0.30	0.30
9	866254	1072896	1065411	**1.24**	**1.23**	0.32	0.26	0.26
10	820408	1072896	1065402	**1.31**	**1.30**	0.31	0.24	0.24
11	820392	1072896	1065405	**1.31**	**1.30**	0.28	0.21	0.22
12	820397	832492	1065398	**1.01**	**1.30**	0.26	0.25	0.20
JPEG Decoder (12 Pages)								
3	1784064	1784064	2052864	**1.00**	**1.15**	0.53	0.53	0.46
4	1105920	1533696	1533696	**1.39**	**1.39**	0.64	0.46	0.46
5	1103104	1103104	1763328	**1.00**	**1.60**	0.52	0.52	0.32
6	1063424	1063553	1063424	**1.00**	**1.00**	0.45	0.45	0.45
7	1054720	1054720	1054720	**1.00**	**1.00**	0.39	0.39	0.39
8	820736	820736	820736	**1.00**	**1.00**	0.43	0.43	0.43
9	820736	820736	820736	**1.00**	**1.00**	0.39	0.39	0.39
10	820736	820736	820736	**1.00**	**1.00**	0.35	0.35	0.35
11	820736	820736	820736	**1.00**	**1.00**	0.32	0.32	0.32

Table 4.1: Summary of application performance on an idealized array

Chapter 5

Resource Allocation

This chapter focuses on the overhead of array reconfiguration. For virtualized paged application execution to be efficient, the scheduler must contain reconfiguration overheads by carefully managing the two key operations of resource allocation and buffer sizing.

5.1 Resource Allocation

5.1.1 Analysis

Given a temporally partitioned graph, the scheduler must for each temporal partition allocate the following: (1) a CP for each page, (2) a memory region in a CMB for each segment and *stitch* buffer, (3) a block of CMB memory to store the state and configuration for each page, segment, and buffer when that object is not active. Recall from Section 2.2 that this work assumes single-ported CMB controllers. Although CMB memory could be large enough to fit several user-specified segments and *stitch* buffers, the single controller constraint implies that only one of them can be active in a given timeslice. This presents an additional restriction for point (2) above, that no two segments or buffers in a temporal partition can share a CMB.

Section 2.3.1 developed an analytical model to relate execution time to underlying architecture parameters and buffer allocation strategies. This model from Equation 2.10 is repeated below for the reader's convenience:

$$T_{run} = K_{max} \sum_{1 \leq j \leq S} \left\{ \max_{i \in ts(j)} (p_i) + \frac{V}{Q} \left[\sum_{i \in ts(j)} \left[\frac{2Qb_i}{W_{io}} f(\overline{B}, L) \right] + s(P, Y) C_{cp} \right] \right\} \tag{5.1}$$

Consider the key factors that affect reconfiguration overhead. The first is the buffer scaling factor Q, which determines the sizes of allocated buffers. The method to select optimal Q is discussed in Section 5.2. However, this section assumes that Q value is set and discusses a resource allocation algorithm that maps compute graph nodes onto CPs and CMBs. The quality of this algorithm translates into the parameters $f(\overline{B}, L)$ and $s(P, Y)$.

These parameters are associated with two competing overheads: off-chip transfer overhead and array reconfiguration overhead—the first and second terms in the square brackets above. The first parameter $0 \leq f(\overline{B}, L) \leq 1$ represents a fraction of buffers that on average must spill off chip in a given timeslice. Resource constraints and the scheduler's ability to preserve the spatial locality of buffers determines f. The second parameter $1 \leq s(P, Y) \leq P$ represents the scheduler ability to take advantage of reconfiguration parallelism in a SCORE architecture. To optimize results, a resource allocation algorithm must preserve spatial locality and maximize reconfiguration parallelism.

Spatial Locality. Temporal partitioning cuts the original computation graph into subgraphs that fit on available hardware. Although each compute page in the original graph appears in *exactly one* of the temporal partitions, the *stitch* segments, inserted between the partitions, reside in two temporal partitions that are not necessarily adjacent in time. To eliminate the need to spill buffer contents, the memory allocation algorithm must allocate for each stitch segment a single CMB memory region throughout the entire application execution.

For example, consider a buffer that owns a CMB memory block in timeslices 1 and 3. The scheduler ideally avoids allocating the same or an overlapping block to any other buffer in timeslices 2, 4 or others. When the memory block is shared between two or more buffers or user segments, the scheduler must transfer buffer contents to and from primary memory between timeslices. Reducing the fraction of buffers that are spilled, $f(\overline{B}, L)$, allows the system to reduce off-chip memory transfer overhead.

For the purposes of CMB memory allocation, blocks with CP state and configuration bitstreams are treated similarly to stitch segments, and therefore the same optimization applies to them.

Reconfiguration Parallelism. Distributed CMBs enable parallel CP reconfiguration. In a system with Y CMBs, up to Y physical compute pages can be configured in parallel. The resource allocation algorithm has a goal of maximizing reconfiguration parallelism by evenly distributing blocks with state and configuration bitstreams among CMBs. This effectively increases the number of CMBs that can be used as sources of reconfiguration (Y), which in turn reduces $s(P, Y)$ and the array reconfiguration overhead.

5.1.2 Resource Allocation Algorithms

Given temporal graph partitioning $\mathcal{P}_n[1..ts]$, the resource allocation module of the static schedule generator first inserts *stitch* buffers. A *stitch* buffer replaces a stream that crosses a temporal partition boundary and stores the intermediate computation state that results from time-multiplexed application execution. The buffer operates in two modes: as a SINK buffer when co-resident with the stream's source page and as a SOURCE buffer when co-resident with the stream's sink page.

```
Function ADD_REQ_STITCHES ($\mathcal{P}_n[], G_n = (V_n, E_n)$)

for all $e \in E_n$ do
  if $src(e) \in \mathcal{P}_n[k] \wedge snk(e) \in \mathcal{P}_n[m] \wedge k \neq m$ then
    # this stitch crosses the timeslice boundary
    # It acts as a sink for data in partition $k$ and a source in partition $m$
    ADD_STITCH_BUFFER($\mathcal{P}_n[k]$, mode = SINK)
    ADD_STITCH_BUFFER($\mathcal{P}_n[m]$, mode = SRC)
  end if
end for
```

The resource allocation algorithm uses partitions $\mathcal{P}_n$ with *stitch* buffers to perform resource allocation. The resource mappings are emitted as two arrays: $CP[ts][k]$ and $CMB[ts][k][a..b]$. The first one $CP[ts][k]$ contains an entry for the compute node residing in CP_k in timeslice ts. The second array $CMB[ts][k][a..b]$ contains an entry for a memory block that resides in CMB_k in address range $(a..b)$ in timeslice ts. Refer to Figure 4.2 for a detailed description of these data structures.

The entries in $CP[]$ and $CMB[]$ arrays can be *locked*, which indicates to the algorithm that the entry cannot be evicted—it must be resident according to the temporal partitioning. CMB memory entries can also be of two types: $segment$ and $config$. The first type $segment$ indicates that the memory block is allocated for a user-specified segment or a stitch segment. The second type $config$ indicates that the memory block contains the CP state and configuration bit-stream.

The algorithm iterates through $\mathcal{P}_n$ and searches for available physical resources. The allocation process consists of two steps. The first step attempts to preserve spatial locality of references to memory. Recall that each compute page appears in *exactly one* of the timeslices. The *stitch* segments, inserted between the partitions, however, reside in two timeslices that are not necessarily adjacent in time. Lines 10–47 of the function PERFORM_RES_ALLOC strive to allocate the same memory block for a *stitch* segment in both timeslices.

At the start of each iteration through $\mathcal{P}_n$, the algorithm copies the contents of CPs and CMBs from the previous timeslice to the current timeslice (lines 10–

19). The algorithm unlocks all entries to make resources available for pages and segments in the current timeslice. "Step 1" iterates through all compute nodes and reclaims any resource that can be reused from a previous timeslice (lines 21–47). The compute pages and memory segments are treated separately. To reclaim a CP the algorithm simply locks its entry. Reclaiming CMB space is complicated by the fact that only one segment or *stitch* buffer can be active in a CMB in one timeslice. Before locking a CMB entry, the algorithm verifies that no other buffer has already locked a CMB (line 35). The pages and segments that were either not allocated previously, did not survive from a previous timeslice, or could not be locked, are placed in the $NodesWOLocations$ list.

"Step 2" of the algorithm allocates physical compute pages and memory blocks for all members of $NodesWOLocations$ (lines 49–66). Two special functions FIND_AVAIL_CP and FIND_AVAIL_CMB, described later in this section, identify unlocked and available physical resources. These resources are added to $CP[]$ and $CMB[]$ data structures and the corresponding entries are locked to prevent subsequent iterations of the algorithm from evicting them.

Steps 3 and 4 of the algorithm were omitted for brevity. They are similar to the first two steps, but allocate CMB memory blocks for CP state and configuration bit-streams. Similar to the first two steps of the algorithm, step 3 attempts to reuse $CMB[]$ array entries from a previous timeslice, and step 4 allocates new entries.

```
1:  Function PERFORM_RES_ALLOC($\mathcal{P}_n, Q$)
2:
3:  $TSCount \Leftarrow |\mathcal{P}_n[]|$ # get number of timeslices
4:  **for all** $ts \in [1..TSCount]$ **do**
5:      # first, the algorithm locks down the nodes that already have locations
6:      # from a previous timeslice. Any node w/o a location will be added to the
7:      # following list to be processed in the Step 2
8:      $NodesWOLocations \Leftarrow \emptyset$
9:
10:     # prepare $CP$ and $CMB$ resource allocation maps for the current timeslice
11:     **if** $ts > 1$ **then** # copy the entries from the prev TS; they could be reused
12:         $\forall_i CP[ts][i] \Leftarrow CP[ts-1][i]$
13:         $\forall_j CMB[ts][j] \Leftarrow CMB[ts-1][j]$
14:     **end if**
15:     # unlock all entries in $ts$ to allow them to be evicted
16:     $\forall_i CP[ts][i].locked \Leftarrow$ **false**
17:     $\forall_j \forall_{(a..b) \in CMB[ts][j]} CMB[ts][j][a..b].locked \Leftarrow$ **false**
18:     # CMB can have 1 active segment: only one segment can be locked
19:     $\forall_j CMBLocks[j] \Leftarrow 0$
```

```

# (Step 1) lock pages and segments still resident from a previous timeslice
**for all** $n \in \mathcal{P}_n[ts]$ **do**
  # check if node $n$ has an entry left over from a previous timeslice
  $found \Leftarrow$ **false**
  **if** IS_PAGE($n$) **then**  # requires a CP
    **for all** $i \in CP[ts][]$ **do**
      **if** $CP[ts][i] = n$ **then**
        $CP[ts][i].locked \Leftarrow$ **true**
        $found \Leftarrow$ **true**
      **end if**
    **end for**
  **else**  # requires a CMB
    **for all** $i \in CMB[ts][]$ **do**
      **for all** $(a..b) \in CMB[ts][i][]$ **do**
        **if** $CMBLocks[i] = 0 \wedge CMB[ts][i][a..b] = n$ **then**
          **if** $CMB[ts][i][a..b].entry\_type = segment$ **then**
            $CMB[ts][i][a..b].locked \Leftarrow$ **true**
            $CMBLocks[i] \Leftarrow 1$
          **end if**
        **end if**
      **end for**
    **end for**
  **end if**
  **if** $found =$ **false then**
    $PUSH(NodeWOLocations, n)$
  **end if**
**end for** # lock pages and segs resident from prev timeslices

# (Step 2) go through nodes w/o locations and find locations for them
**for all** $n \in NodeWOLocations$ **do**
  **if** IS_PAGE($n$) **then**  # requires a CP
    $ind \Leftarrow$ FIND_AVAIL_CP($CP, ts$)
    $CP[ts][ind] \Leftarrow n$
    $CP[ts][ind].locked \Leftarrow$ **true**
  **else**  # requires a CMB
    # compute the size of the mem segment to be allocated:
    # – a user-specified segment has a fixed size
    # – a *stitch* segment size is its computed relative size $b_i \times Q$
    $size \Leftarrow$ BUFFER_SIZE($n, Q$)
```

```
      $[ind, (a..b)] \Leftarrow$ FIND_AVAIL_CMB($CMB, CMBLocks, ts, size, segment$)
      $CMB[ts][ind][a..b] \Leftarrow n$
      $CMB[ts][ind][a..b].entry\_type \Leftarrow segment$
      $CMB[ts][ind][a..b].locked \Leftarrow$ **true**
      $CMBLocks[i] \Leftarrow 1$
    **end if**
  **end for** # iterate through $NodeWOLocations$

  # **Steps 3 and 4** are essentially the same as the steps 1 and 2. They are not
  # shown for brevity. Instead of allocating CPs or CMB space for compute
  # pages and memory segments, Steps 3 and 4 allocate CMB space for CP
  #  state, *i.e.* configuration bitstream, register and FIFO content. The size of
  # the CP state memory block is determined by architecture.

**end for** # iterate through all timeslices

**return** ($CP[], CMB[]$)
```

Functions FIND_AVAIL_CP and FIND_AVAIL_CMB search and identify available physical resources to allocate for compute pages, memory segments, and memory blocks for CP state and configuration bit-stream. The function to find an available CP strives to find an unoccupied physical compute page (lines 6, 10–12). If all pages are occupied then an unlocked CP is returned. The algorithm runs in time $O(P)$, where P is the number of CPs on the reconfigurable array.

```
 1: Function FIND_AVAIL_CP($CMB, ts$)
 2:
 3: $CPCount \Leftarrow |CMB[ts][]|$
 4: # try to find an empty (unused) CP, if none exist, return an unlocked one
 5: **for all** $i \in [1..CPCount]$ **do**
 6:   **if** $CP[ts][i] \neq \emptyset$ **then**  # empty CP
 7:     **if** $CP[ts][i].locked =$ **false then**
 8:       $UnlockedCP \Leftarrow i$
 9:     **end if**
10:   **else**  # found an empty CP, use it
11:     **return** $i$
12:   **end if**
13: **end for**
14: **return** $UnlockedCP$
```

The operation of the function to find available CMB space is more complicated. The algorithm searches through all CMBs, limited only by the $CMBLocks[i]$ array that indicates the number of segments and *stitch* buffers locked in CMB_i. This number cannot exceed one because the CMBs are single-ported. Therefore, if FIND_AVAIL_CMB is searching for CMB space for a segment or a *stitch* buffer, only CMBs with $CMBLocks[i] = 0$ can be considered as candidates.

For every CMB, function FIND_FREE_SPACE returns a memory range $(a..b)$ and an eviction cost c. The eviction cost is a simple metric that indicates the number of bytes of data to be spilled off-chip to free up a memory block of the specified size. If no entry is evicted function FIND_FREE_SPACE returns $c = 0$. If one or more entries is evicted then FIND_FREE_SPACE exhaustively considers all evictions (only *unlocked* entries) that result in a contiguous memory block and returns the lowest cost. If no eviction yields a memory block of the required size, the function returns the special invalid cost. The exhaustive examination is possible in this case because only adjacent entries are considered, and the number of analyzed evictions is the square of the number of entries in the worst case where all entries are unlocked.

The function also computes CMB memory utilization for each CMB. It is a value ranging from 0 to 1 that describes the fraction of a CMB is utilized by locked entries. The memory utilization acts as a tie breaker in case the eviction cost for two or more CMBs is the same. The use of memory utilization in this algorithm is the key behind the scheduler's ability to exploit reconfiguration parallelism. By selecting the least loaded CMB, the scheduler distributes the memory blocks with CP state and configuration bitstreams evenly among CMBs. This enables the scheduler to configure multiple CPs in parallel from multiple single-ported CMBs.

FIND_AVAIL_CMB is a greedy heuristic based on the bin-packing First Fit Decreasing algorithm with the special added restriction of a single active buffer per CMB [GJ79]. FIND_AVAIL_CMB makes the same number of iterations as the number of CMBs, and each iteration invokes two additional routines. The first one, FIND_FREE_SPACE, considers every sequence of adjacent entries for eviction. Because a CMB contains at most one entry from each timeslice, it has a running time of $O(S^2)$, where S is the timeslice count. The second function, COMPUTE_UTILIZATION, adds up the space consumed by the entries. It executes in time $O(S)$. Therefore, the FIND_AVAIL_CMB runs in time $O(Y \times S^2)$.

```
Function FIND_AVAIL_CMB($CMB, CMBLocks, ts, size, entry\_type$)

$cost \Leftarrow cmbsize + 1$ # invalid cost
$cmbcount \Leftarrow |CMB[ts][]|$
$util \Leftarrow 2.0$ # invalid utilization [0,1]
$cmbindex \Leftarrow -1$

```

```
# look at all CMBs and determine whether they are locked, and what would
# be the cost of allocating the specified amount of memory.
for all $i \in [1..cmbcount]$ do
    # allocation attempt can go forward if there are no locks in this CMB,
    # or if the memory for CP context is being allocated
    if $entry\_type = config \vee CMBLocks[i] = 0$ then
        # Find a mem block of the specified size. Cost $c$ is the number of bytes
        # to dump to primary memory to free up space for the requested block.
        $[(a..b), c] \Leftarrow$ FIND_FREE_SPACE($CMB[ts][i], size$)
        $u \Leftarrow$ COMPUTE_UTILIZATION($CMB[ts][i]$)
        # invalid cost indicates that allocation of the requested size is not possible
        if ISVALIDCOST($c$) then
            # find the entry with the min eviction cost and the min utilization
            if $(c < cost) \vee ((c == cost) \wedge (u < util))$ then
                $cost \Leftarrow c$
                $cmbindex \Leftarrow i$
                $range \Leftarrow (a..b)$
                $util \Leftarrow u$
            end if
        end if
    end if
end for
return $[range, cmbindex]$
```

All algorithms presented above are guaranteed to succeed because the temporal partitioner emits subgraphs that “fit” on the array. Note, that the algorithms are greedy and may not optimally minimize off-chip spills or array reconfiguration overhead.

Let us return to the PERFORM_RES_ALLOC to evaluate its running time. The algorithm makes S iterations through all timeslices (line 4), and $P + Y$ iterations in each inner loop. Recall that S is the number of timeslices, P is the number of CPs, and Y is the number of CMBs on the array. The loop that dominates the run time is “Step 2” (lines 49 – 66), which invokes FIND_AVAIL_CMB, the routine with the running time of $O(Y \times S^2)$. Thus, the running time of the entire resource allocation algorithm is $O(S \times (P + Y) \times Y \times S^2)$ or $O(S^3 \times Y \times (P + Y))$.

5.1.3 Evaluation of Algorithms

Two factors determine the quality of the heuristic algorithms described above: (1) how well they leverage the spatial locality in buffer accesses and (2) the amount of reconfiguration parallelism they enable.

Arr Size (CP/CMB)	Wavelet Enc	Wavelet Dec	JPEG Enc	JPEG Dec
2			0.001	
3	0.478		0.018	0.006
4	0.357		0.001	0.006
5	0.387		0.009	0.006
6	0.303	0.832	0.003	0.025
7	0.016	0.780	0.003	0.026
8	0.001	0.743	0.003	0.022
9	0.001	0.636	0.003	0.022
10	0.011	0.597	0.074	0.022
11	0.012	0.397	0	0.045
12	0.001	0.001		0
13	0.001	0.194		0
14	0.002	0.036		0
15	0.002	0.035		0
16	0.009	0.035		

Table 5.1: For each application, the table shows the fraction of live computation state that is moved between CMBs or off chip. CMB size is 256Kbits

Ideally, to leverage the spatial locality, the buffer allocation algorithm must select the same memory region for a buffer for the entire application execution. If that is not possible, the scheduler moves the buffer from one CMB to another at run time. Alternatively, the scheduler spills the contents of a memory block off chip when the block is evicted and restores its contents when the block is needed again.

To consider both of these circumstances together, the reconfigurable array simulator records the size of all memory transfers during application execution. Table 5.1 shows the ratio of the number of bytes the scheduler spills to the total size of the computation intermediate state. In other words, the table shows the fraction of CMB contents moved between CMBs or off chip. The results in the table are intuitive. As the number of CMBs increases, the scheduler moves a smaller fraction of computation state. Note, that only the numbers greater than 0.05 are truly significant. The smaller fractions represent the noise due to anomalies in the heuristic algorithm.

Evaluating the amount of reconfiguration parallelism enabled by the resource allocation algorithm is not trivial. From the analytical model, $C_{array} = \left\lceil \frac{P}{Y} \right\rceil C_{cp}$ is the lower bound on the per time-slice reconfiguration overhead, assuming that all memory blocks with CP state and configuration bit-streams are evenly distributed among Y CMBs. This formulation, unfortunately, is a bit simplistic. There are several circumstances that must be considered.

First, the scheduler configures some CPs only once because they contain the

Arr Size	Wavelet Encoder			Wavelet Decoder		
(CP/CMB)	Seq	Parall	Improv	Seq	Parall	Improv
3	478444	344272	**1.39**			
4	437424	258428	**1.69**			
5	421016	277390	**1.52**			
6	396404	183022	**2.17**	472440	366988	**1.29**
7	330772	107948	**3.06**	415012	262222	**1.58**
8	337876	107296	**3.15**	349380	166916	**2.09**
9	297956	89620	**3.32**	357584	174190	**2.05**
10	265140	86158	**3.08**	345278	168168	**2.05**
11	252514	82050	**3.08**	291952	120242	**2.43**
12	277446	68162	**4.07**	295112	91520	**3.22**
13	269242	64060	**4.20**	193504	64694	**2.99**
14	252834	51760	**4.88**	180878	66916	**2.70**
15	207712	47338	**4.39**	180238	61220	**2.94**
17	214636	38814	**5.53**	178958	48594	**3.68**
19	205152	42602	**4.82**	177678	43864	**4.05**
21	216178	42596	**5.08**	168194	26508	**6.35**
23	219000	42276	**5.18**	158710	26822	**5.92**
25	201312	46698	**4.31**	149226	30924	**4.83**
27	187726	30610	**6.13**	143844	30912	**4.65**
29	178242	26508	**6.72**			
	JPEG Encode			JPEG Decode		
2	300622	104452	**2.88**			
3	249856	96158	**2.60**	122464	56496	**2.17**
4	155654	60024	**2.59**	122464	41968	**2.92**
5	89014	34718	**2.56**	130668	41968	**3.11**
6	101000	38814	**2.60**	101954	29342	**3.47**
7	112666	51120	**2.20**	80804	21778	**3.71**
8	112026	42596	**2.63**	104776	34084	**3.07**
9	111386	51428	**2.17**	104136	30610	**3.40**
10	73828	30610	**2.41**	103496	39128	**2.65**
11	69086	17036	**4.06**	102856	39442	**2.61**
12				69400	8518	**8.15**

Table 5.2: “Seq” column contains total array reconfiguration overhead for a single schedule iteration assuming that all reconfiguration actions occur in sequence, “Parall” column — overhead assuming that non-conflicting reconfiguration actions can occur in parallel, “Improv” column — is the ratio Seq/Parall, *i.e.* parallelism factor.

same virtual compute page throughout the entire application execution. Second, array reconfiguration time C_{array} changes in some cases. For example, the first time a virtual node is placed on a CP, its input FIFO contents do not need to be reloaded. The scheduler must simply initialize page FIFOs. Third, some opportunities for parallel reconfiguration are thwarted by the memory blocks that are evicted off chip. If all memory blocks that contain CP state and bit-streams reside in CMBs, then parallel reconfiguration is possible. If a block ever leaves the chip, it must first be brought back into a CMB for reconfiguration. All accesses to the off-chip memory must be serialized through a single link.

To measure the reconfiguration parallelism that the resource allocation algorithm enables, the static schedule generator computed two values for each schedule. (1) The array reconfiguration overhead over a single schedule iteration, assuming that all configuration actions are performed sequentially. (2) The same overhead if non-conflicting reconfiguration actions are performed in parallel as enabled on the reconfigurable array. These results and the "parallelism" or improvement factor are summarized in Table 5.2.

The table shows that the reconfiguration overhead decreases with array size, suggesting that fewer operations must be performed every schedule iteration. Consider, for example, a compute node that resides in the same CP throughout the application execution. The CP must only be configured once, incurring negligible overhead thereafter. In the table, the parallelism factor increases with the array size, which demonstrates that the resource allocation algorithm takes advantage of additional memory and compute pages as they become available.

5.2 Buffer Sizing

5.2.1 Buffer Sizing Algorithm

The analytical model in Equation 2.10 demonstrates that buffer size is one of the key parameters that determines application execution time. Buffer sizes are directly proportional to the timeslice length and off chip transfer overhead. Careful selection of buffer sizes is important for minimizing application execution time particularly for multirate applications in SCORE.

While the program specifies fixed sizes for user segments such as coefficient lookup tables, the scheduler can freely adjust the size of *stitch* buffers that contain intermediate computation state. The static schedule generator determines the *stitch* segment sizes by performing two operations: computing *relative* buffer sizes for each inter-timeslice stream and computing *absolute* buffer sizes that minimize application execution time.

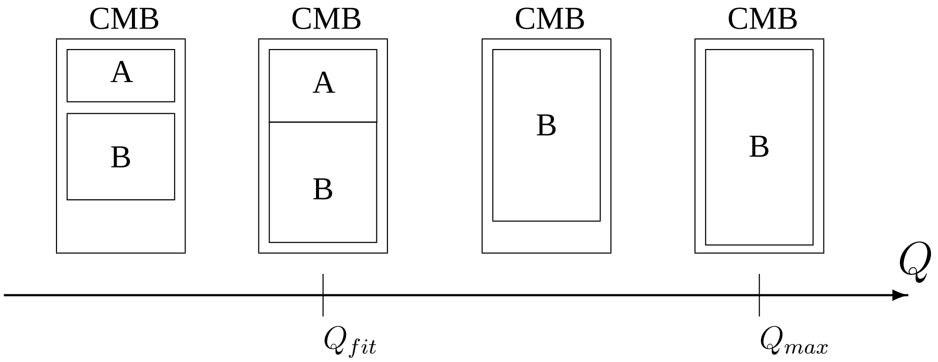


Figure 5.1: Continuum of Q values

Computing *Relative* Buffer Sizes As in the case of temporal partitioning, this work assumes that profiled stream rates are static. The scheduler uses previous work in synchronous data-flow (SDF [BML96]) to efficiently compute relative buffer sizes using a balance equation such as

$$\forall_{i \neq j} \left(\frac{p_i}{b_i} = \frac{p_j}{b_j} \right) \bigwedge \min_i b_i = 1 \tag{5.2}$$

where $p_i \in (0, 1]$ is rate of stream i, and $b_i \geq 1$ is a relative buffer size of stream i. Static schedule generator computes relative buffer sizes b_i for each *stitch* segment in the temporal partition map $\mathcal{P}_n[1..ts]$. The value of b_i represents the smallest desirable size for buffer i. The subsequent operation scales b_i by a factor Q.

The scheduler computes the relative buffer sizes based on the profiled stream rates p_i, which may not reflect the actual dynamic rates in an application for a particular input data set. This, however, does not pose a correctness problem for the methodology in this work. The SCORE page firing semantics, combined with run-time *bufferlock* resolution, guarantee correct execution. On the other side, application performance could be adversely affected if there is a large discrepancy between the profiled and actual stream rates. The *stall detect* hardware reduces the performance impact of such discrepancies by automatically adjusting the time-slice length (Section 3.4.2).

Computing *Absolute* Buffer Sizes The scheduler scales relative buffer sizes b_i proportionally by a factor Q to obtain absolute buffer sizes $B_i = Q \times b_i$ to be allocated in CMBs. The buffer scaling factor Q determines the actual sizes of input and output buffers for each temporal partition. Therefore, it controls the time-slice size—the amount of work the application performs each scheduling step.

Several values of Q are of particular interest to this work (Figure 5.1):

- Q_{fit}: the largest buffer scaling factor where the entire application state resides on-chip. There is no off-chip traffic.
- Q_{max}: the largest buffer scaling factor such that *each buffer fits* in a CMB. $\forall i(Q_{max}b_i \leq L)$
- Q_{opt}: the optimal buffer scaling factor that minimizes total execution time.
- Q_{pred}: the buffer scaling factor the static schedule generator predicts to be optimal, *i.e.* predicted Q_{opt}.
- Q_{best}: the buffer scaling factor that yields the shortest application execution time on a full, cycle-by-cycle simulator (empirical Q_{opt}).

How does the scheduler determine the value of Q_{opt}? Ideally, the scheduler could predict Q_{opt} from the analytical model. However, the resource allocation algorithm makes it difficult to use the model directly. The SCORE resource allocation problem to minimize total execution time is a NP-hard optimization problem even without the multiple constraints of single-ported CMBs, reconfiguration resource conflicts and atomic buffer spills. No exact polynomial time solution to the resource allocation problem exists.

The resource allocation algorithm described in the previous section is a heuristic that approximates the optimal solution. It yields results that can be evaluated in terms of CMB packing efficiency and buffer load balance between CMBs. However, these results are not always monotonic in parameters such as buffer, array and CMB sizes. Therefore, it is difficult to accurately estimate average buffer size $\overline{B}$, the fraction of buffers spilled off chip $f(\overline{B}, L)$, and array reconfiguration parallelism factor $s(P, Y)$. This leads to inaccurate estimates for off-chip swap overhead C_{swap} and array reconfiguration overhead C_{array}, making it difficult to estimate application execution time for a given buffer scaling factor Q. The reverse operation to compute Q_{opt} from the execution time using the analytical model is also difficult.

Rather than using the model, the static schedule generator searches for Q_{opt} by estimating with an abstract simulation the total application execution time for various values of Q. The algorithm searches for the buffer scaling factor that results in the minimum application execution time. The Q search space initially ranges from 1 to Q_{max}. The algorithm employs a hierarchical search strategy, evaluating E values of Q at every hierarchy level. The algorithm chooses the value of Q that yields the smallest expected application execution time as the center point for the subsequent, smaller search range. The process is repeated until the search step size reduces to one.

The hierarchical search algorithm is shown below. The pseudocode in lines 15–25 estimates application performance. Given a value of the buffer scaling factor, in this case the value of tmp, the resource allocation is performed with appropriately scaled *stitch* buffers. Refer to the previous section for details on PER-

FORM_RES_ALLOC. MAKE_RECONF_SCRIPT extracts the script of reconfiguration commands from the resource allocation maps $(CP[], CMB[])$. It then arranges the reconfiguration actions to minimize resource conflicts and maximize parallel CP reconfiguration on the array. This is accomplished with an algorithm based on resource constrained task scheduling that prioritizes tasks on the length of their critical paths.

To complete the process, EXEC_SCRIPT imitates the reconfigurable array controller to compute the cycle cost of executing all script commands—the overhead of array configuration. It performs an abstract, dataless simulation of the script. Unlike the full run-time cycle level simulation, which accounts for buffers that may be partially full, the abstract simulation makes the conservative assumption that all buffers are full. The total application execution time $totalTime$ is the sum of the reconfiguration script overhead obtained from the abstract simulation and the ideal application execution time computed based on profiled stream rates and the graph partitioning.

```
Function QOPT_HIER_SEARCH($\mathcal{P}_n[1..ts], E$)

$Q_{max} \Leftarrow \max(Q)$s.t.$\forall i Q b_i \leq L$

$start \Leftarrow 1$
$end \Leftarrow Q_{max}$
$step \Leftarrow \max((end - start)/E, 1)$
$halfStep \Leftarrow step/2$
$OptVal \Leftarrow \infty$
$Q_{opt} \Leftarrow 0$

while $step > 0$ do  # keep iterate until the search range is tiny
  $tmp \Leftarrow start + halfStep$
  while $tmp \leq end$ do
    # at this point, the required stitch segments have been inserted
    # in $\mathcal{P}_n[]$ and relative buffer sizes $b_i$ have been computed.
    # Perform resource allocation with buffer scaling factor of $tmp$.
    $(CP[], CMB[]) \Leftarrow$ PERFORM_RES_ALLOC($\mathcal{P}_n[], Q \leftarrow tmp$)
    # Make a script of reconfig commands based on allocated resources.
    $CmdList \Leftarrow$ MAKE_RECONF_SCRIPT$(CP[], CMB[])$
    # Evaluate the reconfig overhead of the script by abstract simulation.
    $ovhd \Leftarrow$ EXEC_SCRIPT$(CmdList)$
    # compute ideal application execution time
    $T_{ideal} \Leftarrow K_{max} \sum_{1 \leq j \leq S} \max_{i \in ts(j)}(p_i)$
```

```
        $totalTime \Leftarrow T_{ideal} + ovhd$
        **if** $OptVal > totalTime$ **then**
            $OptVal \Leftarrow totalTime$
            $Q_{opt} \Leftarrow tmp$
        **end if**
        $tmp \Leftarrow tmp + step$
    **end while**

    $start \Leftarrow \max(Q_{opt} - halfStep - 1, 1)$
    $end \Leftarrow \min(Q_{opt} + halfStep + 1, Q_{max})$
    $step \Leftarrow (end - start)/E$
    $halfStep \Leftarrow step/2$
**end while**
**return** $Q_{opt}$
```

The hierarchical search algorithm evaluates $E \times \log_E(Q_{max})$ candidate values of Q, where E is the number of points examined at each level of the search hierarchy. The total running time of QOPT_HIER_SEARCH is thus $O(E \times \log_E(Q_{max}) \times S^3 \times Y \times (P+Y))$, where $O(S^3 \times Y \times (P+Y))$ is the running time of the PERFORM_RES_ALLOC routine. Functions MAKE_RECONF_SCRIPT and EXEC_SCRIPT do not dominate the execution time because they run in time proportional to the array size, $O(P + Y)$.

5.2.2 Quality of Q_{opt} Prediction

Accurately computing Q_{opt} off-line is critical to the quasi-static scheduling methodology. Figure 5.2 shows the wavelet encoder execution time (makespan) with several key values of Q. The graph demonstrates that the predicted Q_{pred} yields a makespan that is close to the empirical Q_{best}—obtained through application execution simulation with schedules constructed with all possible Q values. The data also shows that neither Q_{fit} nor Q_{max} consistently yield the lowest makespan.

Table 5.3 compares application execution times from predicted and empirical optimal values of Q for the four applications. The static schedule generator predicted Q_{pred} yields the application performance that is on average within 10–15% of the ideal. Nevertheless, the quality of Q_{opt} prediction could stand to be improved. The maximum ratio between the makespan obtained from the predicted Q_{pred} and the empirical Q_{best} reaches as high as 1.25. This error is significant particularly on smaller reconfigurable arrays where, due to severely limited on-chip memory capacity, the true relationship between the total application time and Q is difficult to predict accurately.

Arr Size	Wavelet Encoder			Wavelet Decoder		
CP/CMB	MS_{pred}	pred/best	MS_{best}	MS_{pred}	pred/best	MS_{best}
3	2038242	**1.20**	1692538			
4	1904188	**1.18**	1615710			
5	1628052	**1.12**	1455426			
6	963862	**1.17**	823543	1320906	**1.15**	1148372
7	586130	**1.02**	572392	1101414	**1.15**	960510
8	581644	**1.08**	538218	1270218	**1.02**	1240614
9	495306	**1.03**	481138	1885624	**1.01**	1858326
10	384280	**1.32**	291768	748382	**1.06**	704838
11	365600	**1.25**	291498	811053	**1.13**	717373
12	341016	**1.19**	287174	829864	**1.24**	668672
13	286085	**1.00**	286085	543812	**1.01**	540876
14	275472	**1.00**	275114	322602	**1.05**	306770
16	272508	**1.00**	272504	312021	**1.03**	301962
18	299272	**1.10**	272481	301976	**1.03**	292126
20	272504	**1.00**	272500	291921	**1.00**	291921
22	272518	**1.00**	272514	315686	**1.03**	305846
24	268449	**1.00**	268178	314687	**1.03**	305295
26	267834	**1.00**	267834	617463	**1.46**	422391
28	265674	**1.00**	265662			
30	263308	**1.00**	263308			
	JPEG Encoder			JPEG Decoder		
2	3331761	**1.03**	3243353			
3	3220591	**1.02**	3158453	2107097	**1.00**	2106881
4	2173524	**1.01**	2155437	1823395	**1.00**	1823267
5	1559881	**1.00**	1559875	1536189	**1.00**	1535269
6	1150135	**1.00**	1148911	1459220	**1.19**	1228986
7	1174802	**1.01**	1160557	1206113	**1.08**	1117881
8	1157157	**1.00**	1157151	859299	**1.04**	829934
9	1157080	**1.01**	1146979	857409	**1.03**	829207
10	849899	**1.01**	840686	857512	**1.03**	829373
11	786492	**1.00**	786492	829153	**1.00**	829135

Table 5.3: Comparison between makespan (total application execution time) obtained from the predicted Q_{pred} versus the makespan from the empirically obtained Q_{best}.

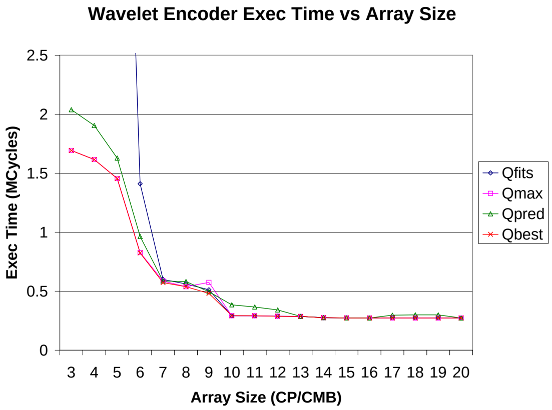


Figure 5.2: Makespan that results from selecting different values for Q.

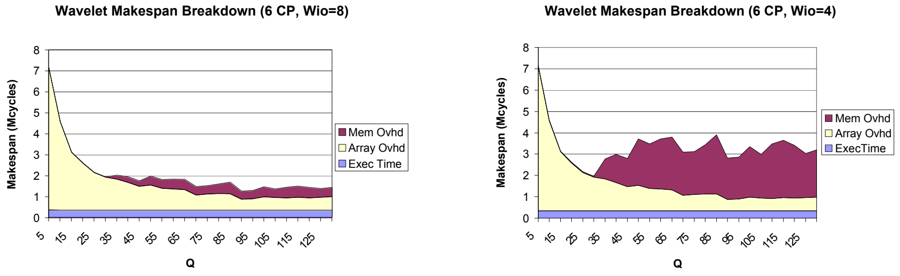
5.3 Evaluation

Section 2.3.1 develops the analytical model that relates execution time T_{run}, architecture parameters, and buffer scaling factor Q. Section 5.3.1 provides simulation results to validate the model. The results demonstrate that the model and the quasi-static scheduler system can adapt to variations in parameters such as off-chip memory bandwidth and CMB size. It is crucial that a robust scheduling system adapts to these parameters to be able to scale with device generations.

The analytical model demonstrates that a multi-rate application implemented with uniform size buffers under-utilizes on-chip memory. Section 5.3.2 quantifies the performance penalty from this memory under-utilization.

5.3.1 Model Operation

Consider variations in off-chip memory bandwidth. Figure 5.3 shows the simulated application execution time breakdown into the off-chip memory transfer overhead, array reconfiguration overhead, and the ideal execution time for various values of Q. The application is a wavelet encoder running on a 6 CP/CMB array. The two graphs represent the simulation results with off-chip memory bandwidths W_{io} of 8 and 4 bits per cycle, respectively. Both share common points for $Q \leq 35$ because the buffers are small enough that the live application state fits in CMBs. This virtually eliminates off-chip traffic. As Q increases further, the run times diverge due to the relative impact of off-chip memory bandwidth.



(a) Array overhead dominates: $Q_{opt} = 95$.

(b) Memory transfers dominate: $Q_{opt} = 35$.

Figure 5.3: Wavelet encoder simulated execution time breakdown into off-chip memory transfer overhead ($\propto 1/W_{io}$), array management overhead, and ideal execution time.

The execution time in Figure 5.3a, corresponding to off-chip bandwidth of 8 bits per second, continues to decrease past $Q = 35$ because the array reconfiguration overhead dominates the execution time. The timeslice lengths increase with Q and help amortize the overhead. The optimal value of the buffer scaling factor is 95. Interestingly, $Q_{opt} = 95$ is larger than $Q_{fit} = 35$. Q_{fit} is the largest Q where the entire application state resides on-chip and for which off-chip traffic is negligible. Notice that extra overhead from spills does not adversely affect the application performance when off-chip bandwidth is abundant with respect to the size of the live state. Although running an application with $Q = Q_{fit}$ minimizes off-chip traffic, it does not always result in the shortest application run time. This is particularly true on smaller arrays with limited memory capacity as in this example with a 6 CP/CMB array.

Figure 5.3b shows the opposite trend for a system with limited off-chip bandwidth of 4 bits per cycle. The execution time increases as Q grows past $Q_{fit} = 35$. Off-chip memory transfers dominate the run-time overhead. Smaller values of Q minimize execution time by permitting live computation state to reside on chip. As both graphs show, in high virtualization on small arrays, the reconfiguration overheads dominate the execution time if not carefully managed.

Table 5.4 provides wavelet encoder execution times for several zones of operation. Refer to Appendix B.3 for complete results for all applications. The table is divided into six zones: high and low off-chip bandwidths for each of the three CMB memory sizes. For each zone, the table contains:

- MS_{fix}: the total execution time with the uniform buffer size schedule
- MS_{var}: the total execution time with the variable buffer size schedule
- *fix/var* : the ratio MS_{fix}/MS_{var}

	Offchip Memory BW 8 bits per cycle					Offchip Memory BW 4 bits per cycle				
CP	MS_{fix}	fix/var	MS_{var}	Q_{opt}	B_{spill}	MS_{fix}	fix/var	MS_{var}	Q_{opt}	B_{spill}
CMB Size (L): 128 Kbits										
3	5216132	**1.55**	3361264	56	139118	12148828	**1.58**	7667522	56	139101
4	4714690	**1.51**	3118522	56	152212	11264519	**1.58**	7139960	56	152202
5	4450734	**1.70**	2613368	64	165952	10957180	**1.69**	6468970	64	165937
6	2844115	**1.51**	1886421	79	241299	7449484	**1.52**	4891488	77	235570
7	2045417	**1.59**	1287266	95	250144	5653968	**1.86**	3039555	96	239953
8	2110170	**1.89**	1113940	97	191532	5432235	**2.22**	2451374	95	180803
9	1822222	**2.01**	906200	150	199424	4411124	**2.21**	1993188	153	179936
10	764194	**1.39**	548422	1989	223317	1721746	**2.46**	698994	505	0
11	806596	**1.74**	463235	1989	129525	1801941	**3.63**	496646	995	0
13	513147	**1.50**	341763	170	0	568007	**1.66**	341763	170	0
15	276170	**1.01**	272498	331	0	278754	**1.02**	272498	331	0
19	272229	**1.00**	272494	8119	0	272229	**1.00**	272496	8119	0
CMB Size (L): 256 Kbits										
3	2267062	**1.34**	1692538	113	107986	4795857	**1.43**	3357634	113	107822
4	2153736	**1.33**	1615710	113	107224	4492198	**1.47**	3053244	113	107047
5	2009326	**1.38**	1455426	130	114178	4437910	**1.62**	2733989	130	113969
6	1238105	**1.50**	823543	194	133935	2637761	**1.87**	1410530	46	0
7	1017004	**1.78**	572392	129	0	2151584	**3.76**	572392	129	0
8	981699	**1.82**	538218	194	31974	1680184	**3.07**	548036	150	0
9	773102	**1.61**	481138	240	0	884999	**1.84**	481140	237	0
10	387984	**1.33**	291768	4051	0	412150	**1.41**	291770	4051	0
11	389162	**1.34**	291498	4051	0	400499	**1.37**	291498	4051	0
13	358524	**1.25**	286085	335	0	361836	**1.26**	286081	335	0
15	273379	**1.00**	272498	331	0	273379	**1.00**	272498	331	0
19	272229	**1.00**	272494	8101	0	272229	**1.00**	272496	14105	0
CMB Size (L): 512 Kbits										
3	1211836	**1.20**	1012716	182	43213	1530190	**1.49**	1025704	116	0
4	1084009	**1.30**	832682	178	20305	1412382	**1.55**	910469	120	0
5	1030598	**1.33**	776253	198	14565	1316788	**1.53**	858749	198	14565
6	646156	**1.60**	403171	389	42278	746448	**1.80**	415114	389	43814
7	515809	**1.45**	356634	391	16704	609932	**1.70**	359656	391	16704
8	566776	**1.68**	338086	389	16128	636002	**1.85**	343034	389	14848
9	508438	**1.61**	316004	613	18752	530000	**1.65**	322134	613	17216
10	291648	**1.00**	292084	4071	0	291648	**1.00**	291658	4951	0
11	291408	**1.00**	290994	4071	0	291406	**1.00**	291264	5556	0
13	286996	**1.00**	285600	457	0	286996	**1.00**	285632	457	0
15	273379	**1.00**	272518	330	0	273379	**1.00**	272498	330	0
19	272234	**1.00**	272496	21321	0	272237	**1.00**	272496	21321	0

Table 5.4: Wavelet Encoder (30 pages): Six operating zones for a SCORE system and application performance comparison with uniform fixed buffer size allocation.

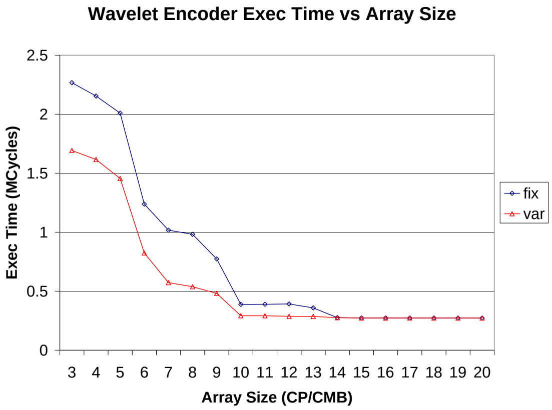


Figure 5.4: Comparison of application run time with variable vs uniform size buffers (CMB size is 256Kbits).

- Q_{opt}: the buffer scaling parameter that yields the execution time MS_{var}
- B_{spill}: the number of bytes spilled off chip during application execution.

The subsequent section analyzes the results for MS_{fix} and *fix/var*. This section describes the way the system adapts to changes in the underlying architecture parameters.

Consider the application execution time on an array with small 128Kbit CMBs and compare the corresponding values of Q_{opt} for every array size. For example, for a 10 CP array, Q_{opt} is 1989 and 505 for off-chip bandwidths of 8 and 4 bits per cycle respectively. On the array with higher off-chip bandwidth, the scheduler uses larger buffers (Q_{opt}) and spills more of the intermediate computation state (B_{spill}). In contrast, for the array with low off-chip bandwidth, the scheduler strives to keep more of the intermediate computation state on chip to avoid costly spills.

Compare the results from 128Kbit CMB arrays with the results from larger CMB sizes of 256Kbits and 512Kbits. They reveal that the scheduler spills less of the intermediate computation state off chip (B_{spill}) and uses larger buffers to fill available CMB memory space (Q_{opt}). The general trend that applies to arrays with all CMB sizes is that the scheduler uses larger buffers and spills more of the intermediate computation state on arrays with the higher off-chip bandwidth.

Stitch buffer utilization: CMB Size 256KBits and Offchip BW = 8 bits/cycle

	Wavelet Encoder		Wavelet Decoder		JPEG Encoder		JPEG Decoder	
CP	Fixed	Variable	Fixed	Variable	Fixed	Variable	Fixed	Variable
2					0.48	0.97		
3	0.25	0.91			0.53	0.97	0.33	1.00
4	0.23	0.92			0.21	0.95	0.30	1.00
5	0.27	0.95			0.54	0.97	0.30	1.00
6	0.34	0.96	0.33	0.98	0.56	0.98	1.00	1.00
7	0.31	0.95	0.29	0.72	0.48	0.97	1.00	1.00
8	0.30	0.95	0.13	0.92	0.48	0.98	1.00	1.00
9	0.29	0.93	0.13	0.78	0.48	0.97	1.00	1.00
10	0.72	0.95	0.18	0.80	1.00	1.00	1.00	1.00
11	0.65	0.94	0.13	1.00			1.00	1.00
12	0.43	0.91	0.13	0.76				
13	0.28	0.98	0.20	0.92				

Table 5.5: Comparison of buffer utilization for fixed and variable size buffer schemes. CMB Size is 256KBits and the off-chip bandwidth is 8 bits/cycle

5.3.2 Comparison of Variable vs Uniform Size Buffers

Section 2.3.1 demonstrates that a multi-rate application implemented with uniform size buffers underutilizes on-chip memory. The highest rate buffer causes application execution to stall early, even if other buffers have available space to continue token processing. Figure 5.4 quantifies the loss in application performance that results from allocating uniform size buffers for a wavelet encoder. The graph shows application execution time from two simulations, using optimal uniform size buffers (MS_{fix}) and using variable size buffers (MS_{var}) with the buffer scaling factor Q_{best} .

This work made a special effort to be fair when comparing MS_{fix} versus MS_{var}. Rather than choosing a single uniform buffer size for all architecture parameters, the optimal buffer size that yields the lowest execution time was empirically determined for each point in the architecture parameter space. MS_{fix} is thus the application execution time with the optimal uniform buffer size for the specific array size, CMB memory size, and off-chip bandwidth.

Figure 5.4 demonstrates that allocating uniform size buffers for the wavelet encoder results in increases in execution time ranging from 1.25 to 1.82x compared to Q_{best}. The problem is particularly severe on smaller array sizes with limited on-chip memory, where application makespan almost doubles with the use of uniform sized buffers.

Table 5.5 confirms the hypothesis that multi-rate applications severely underutilize memory when implemented with uniform buffer sizes. The table compares allocated buffer space utilization with uniform size buffers versus variable size buffers. With uniform size buffers, wavelet encoder application utilizes on average only 30–40% of the allocated buffer space. In contrast, variable buffer sizes enable a use of 91–98% of the allocated capacity. The variable buffer size utilization is not 100%. This is due to deviations of the profiled rates from the actual stream rates for an input data-set in a simulation.

Table 5.4 shows the improvements in application performance when variable size rather than uniform fixed size buffers are employed (column *fix/var*). The reductions in execution time from fixed to variable size buffers are not monotonic with the array size due to the instabilities in resource allocation algorithms. These same instabilities impede direct computation of Q_{opt} from the analytical model as discussed in Section 5.2.1.

However, the general trend in application performance improvement is intuitive. As the array size increases and resources become abundant, the ratio between the execution time with fixed and variable buffer sizes decreases. Multi-rate applications scheduled with uniform fixed buffer sizes under-utilize memory. Accordingly, the simulations demonstrate the highest performance improvements on reconfigurable arrays with severely constrained resources—CMB size of 128Kbits and the off-chip memory bandwidth of 4 bits per cycle. On the resource constrained device, the variable size buffer scheme permits the scheduler to pack the buffers the most tightly together and efficiently utilize the limited hardware. Consequently, as the array size scales to larger CMB sizes and higher bandwidths, the improvements in application performance decrease.

Chapter 6

Conclusion

A run-time scheduler plays a key role in supporting the abstractions of SCORE, that gives a programmer expressive power and flexibility for dynamic data dependent computations. A poorly implemented scheduler may drastically diminish the potential performance gains offered by FPGAs and severely limit the model's applicability.

This work presents a taxonomy of scheduling solutions to demonstrate that, in addition to the natural solution of a fully dynamic scheduler, there exists a rich space of solutions of varying complexity, quality, and restrictions on application features. While all solutions preserve the semantic and expressive power of the SCORE compute model, only a subset yields efficient practical implementations.

For efficient page scheduling, any time-multiplexed implementation must contain scheduling and array reconfiguration overheads. To address the scheduling overhead, this work investigates static and dynamic scheduling approaches and demonstrates the quasi-static approach to have superior scheduling quality and substantially lower run-time overhead than the dynamic approach. The quasi-static implementation reduces run-time scheduling overhead by a factor ranging from 5.5 to 9.4 across the applications. This reduced overhead of 10–20 thousand clock cycles per timeslice makes the model efficient on applications with short execution time or rapidly changing run-time behavior. With the reduced runtime scheduling overhead and superior scheduling quality, the quasi-static implementation decreases execution times by a factor ranging from 2.7 to 5.4 for a set of applications containing both static and data-dependent components.

This work focuses on two key scheduler components: temporal graph partitioning and physical resource allocation.

The analytical model that captures the relationship between the execution time and array utilization yields several algorithms for temporal partitioning. These heuristic algorithms, driven by the array utilization metric, deliver results that are on average within 10% of the ideal.

The quasi-static scheduling methodology implements a flexible resource allocation strategy which accommodates the multi-rate and dynamic rate computations typical to many media processing applications. These applications present a challenge to the scheduler to efficiently utilize off-chip bandwidth and on-chip memory. This work demonstrates that to minimize execution time, the scheduler must match buffer sizes with compute graph flow rates, and scale buffers to efficiently balance off-chip spills and array reconfiguration overhead. The quasi-static scheduler implements these principles without restricting the dynamic data dependent execution semantics of the SCORE graphs. It offers a balanced combination of static scheduling techniques with inexpensive hardware mechanism for stall detection.

With variable size buffers and a robust resource allocation scheme, the reduction in the execution time relative to uniform size buffer scheme typically exceeds 50%. The improvements are particularly significant on smaller devices with constrained memory and bandwidth. These results persist over a range of applications and architecture parameters.

The quasi-static scheduling methodology was evaluated with several image processing applications that are the most natural match to the streaming dataflow programming model of SCORE. These applications include a range of components with different behaviors from dynamic dataflow operators such as Huffman coders to static rate operators such as discrete cosine transforms. The performance results obtained with these applications demonstrate the viability and effectiveness of the low overhead, quasi-static scheduling for the applications in this domain.

These image processing applications share several common characteristics that favor the quasi-static scheduling. Although token emission and consumption rates of their many components are dynamic and dependent on the input data, the flow rates do not vary erratically, but average around a particular value. The hardware stall detect mechanism effectively adapts application execution to short term changes in rates of individual operators. It remains an open issue to determine and quantify the effects of using average profiled rates to build static schedules for applications, particularly for those where the profiled rates differ significantly from the actual dynamic rates.

The applications evaluated in this work do not contain large communication dependency cycles in their compute graphs. A large cycle that must be temporally partitioned between timeslices precludes efficient application execution on any virtuallized system, and thus this work did not optimize the scheduler operation for such applications. An analogous case in conventional operating systems occurs whenever an application's "working set" of virtual memory pages is smaller than available physical memory. Applications with large cyclical dependencies cannot be efficiently executed on a SCORE platform with insufficient hardware resources to accommodate a large cycle in the critical path.

This work presents an empirically validated analytical model that describes the relationship between application execution time and buffer sizes. This project demonstrates that the quasi-static scheduler system is robust to the variations in the key architecture parameters such as on-chip memory size and off-chip memory bandwidth. This permits the scheduler to automatically scale application performance across device generations—enabling application longevity and expanding SCORE's applicability.

Appendix A

SCORE Applications

This work evaluates performance of four applications with the quasi-static, fully dynamic, and fully static schedulers. This section provides a short description of these four applications, their compute graph topology and general functionality of individual pages and segments. This is not intended to be a complete documentation for these applications, but rather an attempt to give the reader a glance at the application structure and operation. For the complete documentation, the reader is referred to the applications' author Joseph Yeh.

A.1 JPEG Encoder

Figure A.1 contains the compute graph for a standard compliant JPEG encoder application. The application converts a raw 512×512 pixel monochrome image into a JPEG bitstream.

The JPEG images consist of 8×8 pixel blocks, which are encoded independently. The processor sends 8 image pixels at a time in a row major order. The computation begins with a one dimensional discrete cosine transform in `(0) fllm`, followed by a transpose operator `(1) tpose`, and another one dimensional DCT in `(2) fllm`. Together these three pages implement a two-dimensional discrete cosine block transform, which converts 64 pixels from spatial domain to frequency domain. The frequency coefficients are arranged in such a way that the lowest frequency coefficient is in the upper-left corner of the block, and the highest frequency coefficient is in the bottom-right corner. The zigzag scan `(3) zigzag` turns the two-dimensional data into a one-dimensional stream to be compressed.

Compression begins with quantization, performed by `(4) jquant` operator with the help of the quantization table in segment `(11)`. Then the application performs zero-length encoding of the quantized data in `(5) JZLE`, which emits data for Huffman encoding. There are two sets of Huffman tables in this application. Segments `(13)` and `(12)` contain codewords and codeword lengths for encoding the DC value in the block (the value in the upper-left corner). Segments `(14)` and `(15)` contain codewords and codeword lengths for encoding the higher frequency data. Operator `(8) MixToHuff` routes the data from the JZLE and the Huffman tables to the `(9) HuffMachine` to complete the encoding process.

The last operator `(10) CheckZero` guarantees that the emitted JPEG bitstream is valid. The value 00 is a special start code in the JPEG standard, and `CheckZero` appends FF after each 00 to disable its special meaning.

A.2 JPEG Decoder

Figure A.2 contains the compute graph for the JPEG decoder application, which performs the inverse operation as compared to the encoder. The application consumes a JPEG-compliant bitstream to emit a 512×512 raw monochrome image.

Pages `(9) DecHuff`, `(10) DecSym`, and `(11) ftabmod` decompress the incoming streams using a Huffman table in segment `(15)` and zero-length decoding. The application inverse quantizes the decompressed data stream in `(8) jdquant` using the quantization parameters in segment `(14)`. The application uses two read/write segments `(12)` and `(13)` in alternation to reconstruct the blocks of frequency coefficients that will be converted into spacial domain. While the inverse quantizer is writing to segment `(12)`, operator `(4) distrib` is reading from segment `(13)`, and vice versa. The operators `(6) zigzag` and `(5) read_seg` generate required segment access addresses and control tokens (read or write) to manage concurrent read and write operations.

Page `(4) distrib` forwards the data from segments `(12)` and `(13)` to the two-dimensional inverse DCT. 2D IDCT consists of `(0) illm`, `(1) tpose`, and `(2) illm`, which perform an one-dimensional IDCT, followed by a transpose, and another one-dimensional IDCT to compute the spatial domain block. The last operator `bl` verifies that all emitted data is bound to the valid range between 0 and 255.

A.3 Wavelet-based Encoder

Figure A.3 contains the compute page diagram for the wavelet-based encoder application that compresses 512×512 pixel monochrome image. The application consists of 30 virtual compute pages and 6 user-defined segments.

The application driver code, which executes on the processor, feeds the image into the compute graph one pixel at a time in a row major order. The process-

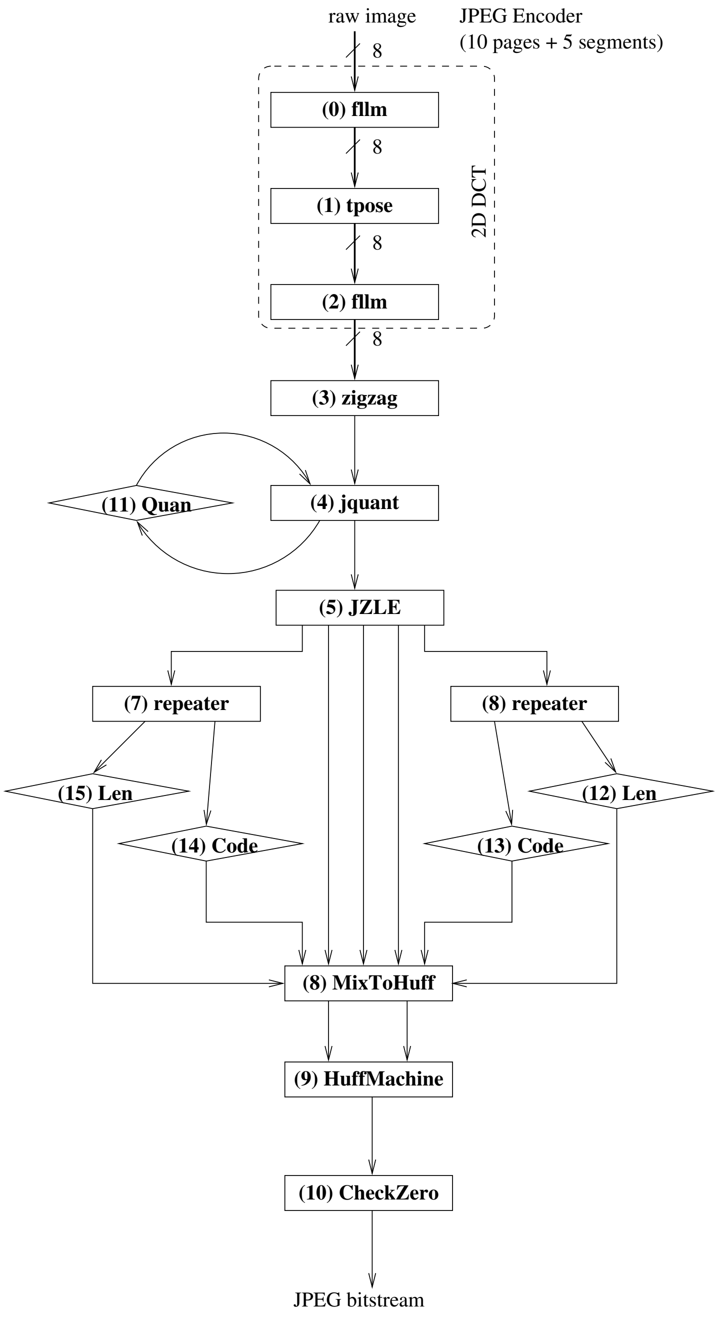


Figure A.1: JPEG Encoder: compute page graph

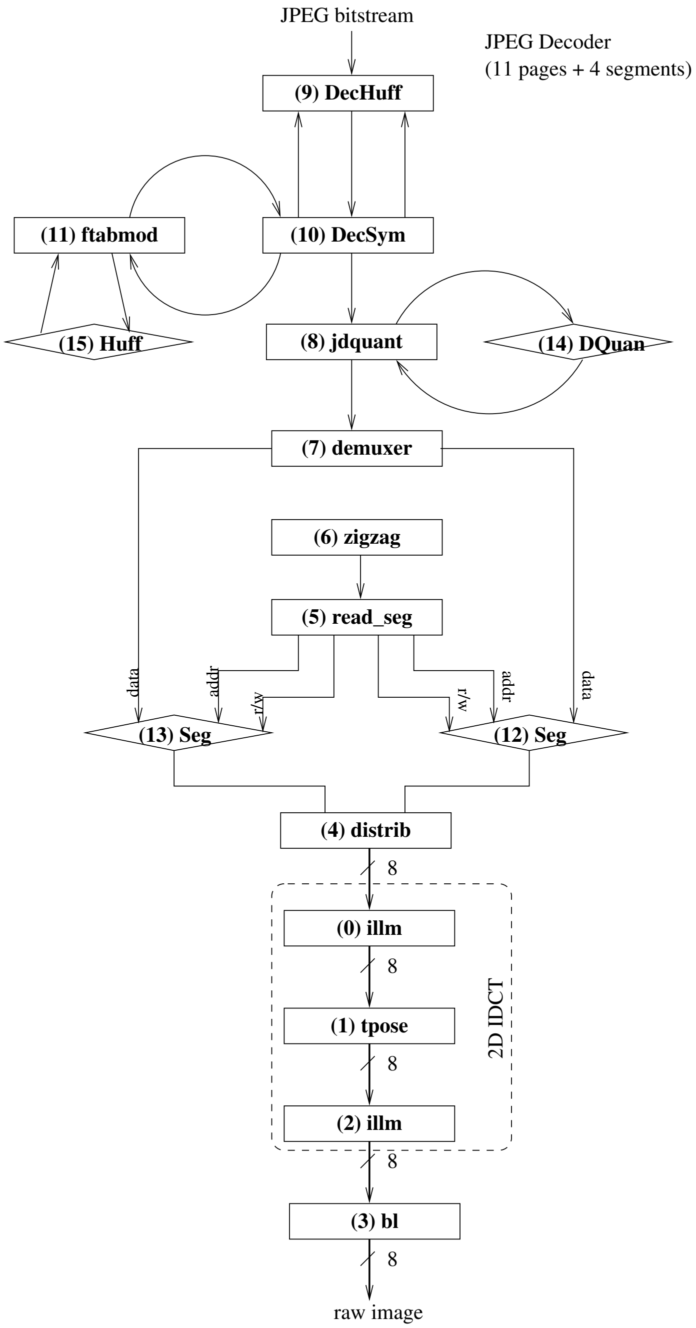


Figure A.2: JPEG Decoder: compute page graph

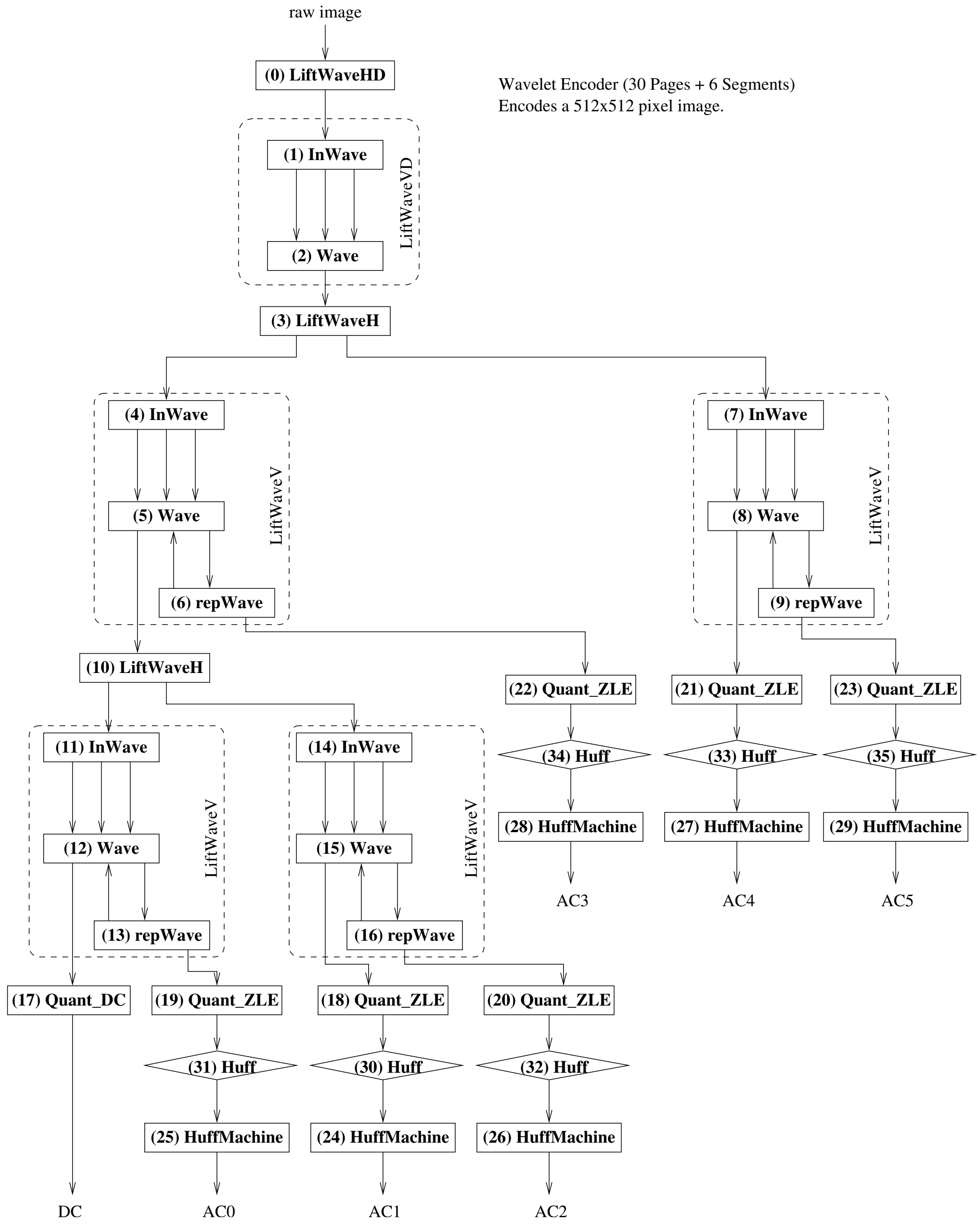


Figure A.3: Wavelet Encoder: compute page graph

ing begins with `(0) LiftWaveHD`, which subsamples the incoming horizontal data stream by 2 after passing it through a low-pass filter. Notice, that high-frequency data is discarded in this first pass. After the transformation on image rows, the application performs the same transformation on image columns using `(1) InWave` and `(2) Wave`. Logically, the `InWave` transposes the data. In practice, the transpose is a costly operation, and `InWave` uses three streams between `(1)` and `(2)` as delay buffers to rearrange the incoming data for `Wave` to apply a FIR filter to image columns. Notice, that the first column pass discarded high-frequency data as well. Together, `LiftWaveHD` and `LiftWaveVD` reduce the original data stream size by a factor of 4.

The application performs two more wavelet passes consisting of `LiftWaveH` and `LiftWaveV` to obtain a DC and six frequency bands (AC0–AC5). These two operations do not discard high frequency data. Before emitting DC, the application quantizes DC data in `(17) Quant_DC`. To compress the data in six frequency bands, the application starts with quantization and zero-length encoding (`Quant_ZLE`). The encoded data is used as an address into a Huffman table (segment `Huff`), which emits the codewords into a `HuffMachine` page, responsible for assembling and communicating the data back to the processor. A compute page `Quant_ZLE` and a Huffman table are unique for each frequency band.

A.4 Wavelet-based Decoder

Figure A.4 contains the compute graph page for the wavelet-based decoder. This application performs the inverse operation than that of wavelet-based encoder. Given a DC-band data stream and six AC-band data streams, the application emits a raw 512×512 monochrome image.

The incoming frequency band data is first decompressed using Huffman tables in segments 27–32 and then using zero-length decoding. After the data is dequantized, the application assembles it back into a single stream by up-sampling and adding appropriate streams together to form the original image. Notice, that `DecWaveV` pages, which reverse the wavelet transformation in the vertical (column) direction, use self-looping streams as delay lines to rearrange the incoming data in row major order for processing in the column major order.

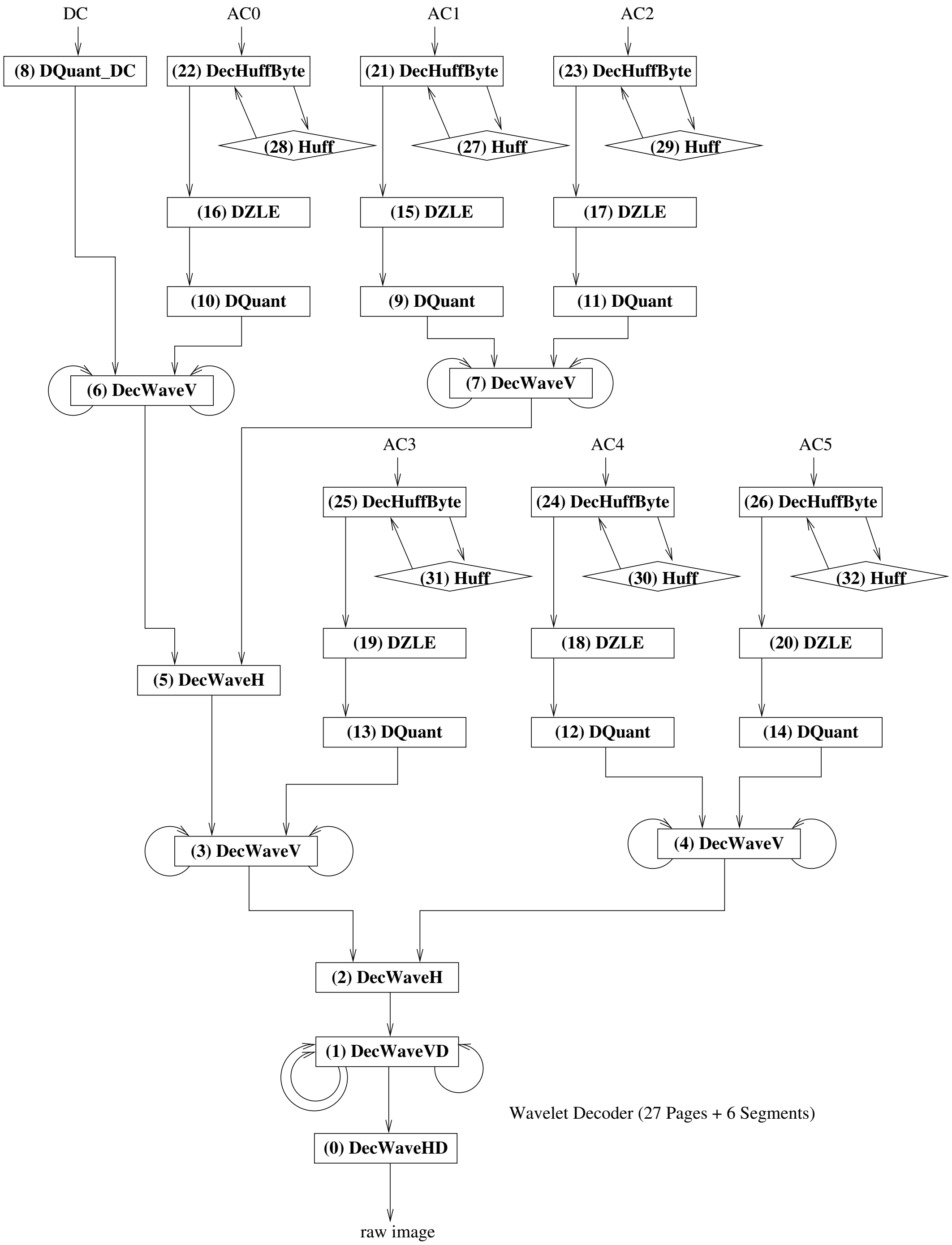


Figure A.4: Wavelet Decoder: compute page graph

Appendix B

Measured Results

B.1 Dynamic vs Quasi-Static Schedulers

Wavelet Encoder (30 pages) : Summary of Total Execution Time (MCycles)

Array Size	Dynamic			Static Real	Quasi-Static			Speedup vs Dyn.	
	Ideal	Real	% Ovhd		Ideal	Real	% Ovhd	Static	Q-Static
6	5.968	7.324	*18.5%*	4.144	0.795	0.859	*7.5%*	**1.77**	**8.52**
7	5.447	6.749	*19.3%*	3.502	0.684	0.756	*9.5%*	**1.93**	**8.92**
8	4.166	5.400	*22.9%*	2.887	0.619	0.683	*9.3%*	**1.87**	**7.91**
9	4.162	5.497	*24.3%*	2.398	0.613	0.662	*7.5%*	**2.29**	**8.30**
10	3.156	4.324	*27.0%*	2.308	0.565	0.630	*10.3%*	**1.87**	**6.86**
11	2.141	3.025	*29.2%*	2.244	0.537	0.591	*9.0%*	**1.35**	**5.12**
12	2.142	3.014	*28.9%*	1.662	0.508	0.560	*9.3%*	**1.81**	**5.38**
13	2.137	3.144	*32.0%*	1.683	0.507	0.568	*10.8%*	**1.87**	**5.53**
14	1.872	2.754	*32.0%*	1.595	0.465	0.513	*9.3%*	**1.73**	**5.37**
15	1.628	2.412	*32.5%*	1.071	0.458	0.492	*6.9%*	**2.25**	**4.91**
16	1.367	2.130	*35.8%*	1.663	0.489	0.541	*9.5%*	**1.28**	**3.94**
17	1.089	1.839	*40.8%*	1.090	0.466	0.514	*9.2%*	**1.69**	**3.58**
18	1.086	1.757	*38.2%*	1.067	0.459	0.503	*8.6%*	**1.65**	**3.50**
19	1.095	1.871	*41.5%*	1.021	0.439	0.474	*7.3%*	**1.83**	**3.95**
20	0.827	1.387	*40.4%*	1.025	0.441	0.487	*9.3%*	**1.35**	**2.85**
21	0.807	1.380	*41.5%*	1.010	0.437	0.479	*8.7%*	**1.37**	**2.88**
22	0.788	1.374	*42.6%*	0.974	0.419	0.452	*7.2%*	**1.41**	**3.04**
23	0.777	1.407	*44.7%*	0.949	0.414	0.445	*7.1%*	**1.48**	**3.16**
24	0.776	1.378	*43.7%*	0.942	0.407	0.461	*11.8%*	**1.46**	**2.99**
25	0.777	1.434	*45.8%*	0.969	0.411	0.450	*8.7%*	**1.48**	**3.19**
26	0.774	1.410	*45.1%*	0.956	0.413	0.453	*8.9%*	**1.47**	**3.12**
27	0.775	1.420	*45.4%*	0.958	0.413	0.452	*8.7%*	**1.48**	**3.14**
28	0.771	1.409	*45.3%*	0.949	0.416	0.471	*11.6%*	**1.48**	**2.99**
29	0.528	1.006	*47.5%*	0.947	0.407	0.456	*10.8%*	**1.06**	**2.21**
30	0.275	0.461	*40.2%*	0.442	0.412	0.433	*4.9%*	**1.04**	**1.06**
Geometric Mean			*35.0%*				*8.7%*	**1.6**	**4.0**

Wavelet Decoder (27 pages) : Summary of Total Execution Time (MCycles)

	Dynamic				Quasi-Static			Speedup vs Dyn.	
Array Size	Ideal	Real	% Ovhd	Static Real	Ideal	Real	% Ovhd	Static	Q-Static
6	9.659	11.631	*17.0%*	3.275	0.812	0.846	*4.0%*	**3.55**	**13.75**
7	8.447	10.319	*18.1%*	2.618	0.735	0.765	*4.0%*	**3.94**	**13.49**
8	7.403	9.137	*19.0%*	2.613	0.721	0.753	*4.3%*	**3.50**	**12.13**
9	5.814	7.356	*21.0%*	2.021	0.697	0.727	*4.1%*	**3.64**	**10.11**
10	5.033	6.489	*22.4%*	1.999	0.663	0.691	*4.1%*	**3.25**	**9.39**
11	5.423	7.165	*24.3%*	1.933	0.614	0.642	*4.3%*	**3.71**	**11.16**
12	6.227	8.187	*23.9%*	1.887	0.608	0.637	*4.5%*	**4.34**	**12.86**
13	5.817	7.844	*25.8%*	1.813	0.594	0.621	*4.3%*	**4.33**	**12.63**
14	5.125	6.887	*25.6%*	1.185	0.530	0.553	*4.1%*	**5.81**	**12.46**
15	4.231	5.816	*27.2%*	1.123	0.511	0.533	*4.1%*	**5.18**	**10.91**
16	2.978	4.156	*28.3%*	1.178	0.533	0.557	*4.3%*	**3.53**	**7.47**
17	3.325	4.621	*28.0%*	1.136	0.525	0.549	*4.3%*	**4.07**	**8.42**
18	1.991	2.803	*29.0%*	1.095	0.511	0.534	*4.3%*	**2.56**	**5.25**
19	2.560	3.600	*28.9%*	1.035	0.496	0.519	*4.4%*	**3.48**	**6.93**
20	2.543	3.683	*30.9%*	1.034	0.497	0.519	*4.4%*	**3.56**	**7.09**
21	2.536	3.689	*31.3%*	1.030	0.497	0.519	*4.3%*	**3.58**	**7.11**
22	2.425	3.573	*32.1%*	1.032	0.499	0.521	*4.2%*	**3.46**	**6.86**
23	0.893	1.418	*37.0%*	1.029	0.498	0.519	*4.2%*	**1.38**	**2.73**
24	1.126	1.717	*34.4%*	1.030	0.499	0.520	*4.1%*	**1.67**	**3.30**
25	0.621	1.068	*41.8%*	1.037	0.509	0.531	*4.2%*	**1.03**	**2.01**
26	0.739	1.167	*36.7%*	1.044	0.517	0.539	*4.0%*	**1.12**	**2.17**
27	0.368	0.526	*30.0%*	0.501	0.475	0.493	*3.7%*	**1.05**	**1.07**
28	0.369	0.536	*31.2%*	0.505	0.479	0.498	*3.7%*	**1.06**	**1.08**
29	0.368	0.566	*34.9%*	0.505	0.479	0.497	*3.7%*	**1.12**	**1.14**
30	0.377	0.533	*29.2%*	0.503	0.477	0.495	*3.7%*	**1.06**	**1.08**
	Geometric Mean		*27.7%*				*4.1%*	**2.60**	**5.43**

JPEG Encoder (13 pages) : Summary of Total Execution Time (MCycles)

Array Size	Dynamic			Static Real	Quasi-Static			Speedup vs Dyn.	
	Ideal	Real	% Ovhd		Ideal	Real	% Ovhd	Static	Q-Static
4	4.832	6.368	*24.1%*	6.756	2.171	2.341	*7.3%*	**0.94**	**2.72**
5	7.134	9.086	*21.5%*	3.460	1.400	1.479	*5.3%*	**2.63**	**6.15**
6	6.458	8.539	*24.4%*	4.682	1.427	1.516	*5.9%*	**1.82**	**5.63**
7	3.875	5.038	*23.1%*	3.212	1.322	1.397	*5.4%*	**1.57**	**3.61**
8	1.655	2.349	*29.6%*	3.173	1.278	1.360	*6.0%*	**0.74**	**1.73**
9	1.672	2.406	*30.5%*	3.216	1.012	1.052	*3.8%*	**0.75**	**2.29**
10	1.635	2.308	*29.2%*	2.179	0.955	0.980	*2.6%*	**1.06**	**2.36**
11	1.635	2.420	*32.4%*	2.132	0.943	0.977	*3.5%*	**1.13**	**2.48**
12	2.621	3.678	*28.7%*	2.194	0.958	0.991	*3.3%*	**1.68**	**3.71**
13	0.797	0.939	*15.1%*	0.899	0.865	0.879	*1.6%*	**1.04**	**1.07**

Geometric Mean *25.3%* *4.1%* **1.23** **2.81**

JPEG Decoder (12 pages) : Summary of Total Execution Time (MCycles)

Array Size	Dynamic			Static Real	Quasi-Static			Speedup vs Dyn.	
	Ideal	Real	% Ovhd		Ideal	Real	% Ovhd	Static	Q-Static
3	10.672	13.312	*19.8%*	6.972	2.836	3.004	*5.6%*	**1.91**	**4.43**
4	5.925	8.010	*26.0%*	4.851	1.580	1.701	*7.1%*	**1.65**	**4.71**
5	5.951	7.580	*21.5%*	4.949	1.666	1.796	*7.2%*	**1.53**	**4.22**
6	3.904	5.077	*23.1%*	3.116	1.290	1.379	*6.5%*	**1.63**	**3.68**
7	1.860	2.589	*28.2%*	1.919	1.153	1.192	*3.3%*	**1.35**	**2.17**
8	1.651	2.196	*24.8%*	2.114	0.946	0.979	*3.4%*	**1.04**	**2.24**
9	1.667	2.260	*26.2%*	2.084	0.942	0.975	*3.4%*	**1.08**	**2.32**
10	1.624	2.225	*27.0%*	2.050	0.940	0.974	*3.5%*	**1.09**	**2.29**
11	1.623	2.518	*35.5%*	2.076	0.933	0.965	*3.4%*	**1.21**	**2.61**
12	0.796	0.910	*12.6%*	0.889	0.856	0.869	*1.5%*	**1.02**	**1.05**

Geometric Mean *23.7%* *4.1%* **1.32** **2.73**

Average Timeslice Overhead (KCycles)

Array Size	Wavelet Encoder			Wavelet Decoder			JPEG Encoder			JPEG Decoder		
	Dynamic	Qstatic	Reduction	Dynamic	Qstatic	Reduction	Dynamic	Qstatic	Reduction	Dynamic	Qstatic	Reduction
3										45.5	7.5	**6.1**
4							57.9	7.5	**7.7**	50.3	6.6	**7.6**
5							56.7	13.4	**4.2**	54.6	7.3	**7.5**
6	52.6	15.3	**3.4**	57.2	7.5	**7.6**	54.6	6.2	**8.8**	56.6	7.2	**7.9**
7	62.2	13.7	**4.5**	68.0	8.0	**8.5**	61.3	7.0	**8.7**	71.9	5.8	**12.4**
8	69.1	15.4	**4.5**	64.0	8.4	**7.6**	105.8	6.5	**16.3**	107.8	15.2	**7.1**
9	76.3	17.7	**4.3**	65.7	10.3	**6.4**	108.0	13.2	**8.2**	110.6	15.8	**7.0**
10	92.6	17.7	**5.2**	66.9	9.8	**6.8**	103.7	11.8	**8.8**	119.5	10.9	**11.0**
11	101.5	14.0	**7.2**	59.8	9.6	**6.2**	106.3	17.2	**6.2**	109.4	14.1	**7.8**
12	97.5	22.0	**4.4**	62.5	9.6	**6.5**	97.5	17.3	**5.6**	87.4	9.1	**9.6**
13	95.4	24.0	**4.0**	81.8	9.0	**9.0**	93.2	9.8	**9.5**			
14	110.2	28.4	**3.9**	63.5	10.4	**6.1**						
15	109.8	22.6	**4.9**	86.6	9.5	**9.1**						
16	110.7	25.1	**4.4**	84.3	9.8	**8.6**						
17	124.0	22.3	**5.6**	74.3	9.9	**7.5**						
18	150.0	24.1	**6.2**	91.5	8.8	**10.4**						
19	144.2	27.9	**5.2**	79.5	7.9	**10.1**						
20	150.4	23.0	**6.6**	87.9	7.6	**11.6**						
21	142.9	21.9	**6.5**	93.4	7.6	**12.3**						
22	147.9	30.7	**4.8**	94.0	7.5	**12.5**						
23	144.3	16.7	**8.7**	89.7	7.2	**12.5**						
24	139.5	16.5	**8.5**	86.7	6.8	**12.8**						
25	129.1	21.7	**6.0**	107.2	7.3	**14.6**						
26	142.9	19.8	**7.2**	103.0	6.9	**15.0**						
27	141.1	12.9	**11.0**	194.2	11.2	**17.3**						
28	129.2	23.2	**5.6**	186.0	11.6	**16.1**						
29	132.3	24.7	**5.4**	166.4	21.9	**7.6**						
30	93.4	18.3	**5.1**	112.0	20.8	**5.4**						
Geometric Mean			**5.5**			**9.4**			**7.9**			**8.2**

B.2 Temporal Partitioning Heuristics

Wavelet Encoder (30 Pages) : Summary of Temporal Partitioning Algorithms

Array Size	Execution Time			Improvement		Measured Array Activity		
	Exhaustive	Topological	Mincut	Topo/Exh	Min/Exh	Exhaustive	Topological	Mincut
6	375808	424704	409856	**1.13**	**1.09**	0.35	0.31	0.32
7	329984	352768	357120	**1.07**	**1.08**	0.34	0.32	0.31
8	311808	330496	319488	**1.06**	**1.02**	0.32	0.30	0.31
9	306944	328192	311040	**1.07**	**1.01**	0.28	0.27	0.28
10	288000	306944	300544	**1.07**	**1.04**	0.27	0.26	0.26
11	287488	292096	297472	**1.02**	**1.03**	0.25	0.24	0.24
12	285184	289280	297472	**1.01**	**1.04**	0.23	0.23	0.22
13	283648	285440	285184	**1.01**	**1.01**	0.21	0.21	0.21
14	273746	278016	285184	**1.02**	**1.04**	0.20	0.20	0.20
15	272640	272640	285184	**1.00**	**1.05**	0.19	0.19	0.18
16	269881	272640	285184	**1.01**	**1.06**	0.18	0.18	0.17
17	268800	272640	285184	**1.01**	**1.06**	0.17	0.17	0.16
18	268800	272640	280832	**1.01**	**1.04**	0.16	0.16	0.16
19	268800	272640	280832	**1.01**	**1.04**	0.15	0.15	0.15
20	268800	272640	280832	**1.01**	**1.04**	0.15	0.14	0.14
21	268800	272640	268800	**1.01**	**1.00**	0.14	0.14	0.14
22	268800	272640	268800	**1.01**	**1.00**	0.13	0.13	0.13
23	268800	268800	268800	**1.00**	**1.00**	0.13	0.13	0.13
24	268544	268800	268800	**1.00**	**1.00**	0.12	0.12	0.12
25	268032	268800	268032	**1.00**	**1.00**	0.12	0.12	0.12
26	266240	268544	266240	**1.01**	**1.00**	0.11	0.11	0.11
27	265728	268032	266240	**1.01**	**1.00**	0.11	0.11	0.11
28	265254	266240	266240	**1.00**	**1.00**	0.11	0.11	0.11
29	265216	266240	266240	**1.00**	**1.00**	0.10	0.10	0.10
30	263424	263424	263424	**1.00**	**1.00**	0.10	0.10	0.10

Wavelet Decoder (27 Pages) : Summary of Temporal Partitioning Algorithms

Array Size	Execution Time			Improvement		Measured Array Activity		
	Exhaustive	Topological	Mincut	Topo/Exh	Min/Exh	Exhaustive	Topological	Mincut
6	315217	373094	333201	**1.18**	**1.06**	0.38	0.32	0.36
7	309841	370021	349857	**1.19**	**1.13**	0.33	0.28	0.30
8	295221	391354	323966	**1.33**	**1.10**	0.31	0.23	0.28
9	289544	461761	340380	**1.59**	**1.18**	0.28	0.17	0.24
10	287008	323136	306231	**1.13**	**1.07**	0.25	0.22	0.24
11	285990	307019	306260	**1.07**	**1.07**	0.23	0.21	0.22
12	286004	332647	306256	**1.16**	**1.07**	0.21	0.18	0.20
13	286004	332659	306273	**1.16**	**1.07**	0.20	0.17	0.18
14	272162	435358	306277	**1.60**	**1.13**	0.19	0.12	0.17
15	272162	298822	306281	**1.10**	**1.13**	0.18	0.16	0.16
16	269600	298824	306293	**1.11**	**1.14**	0.17	0.15	0.15
17	269600	313933	339062	**1.16**	**1.26**	0.16	0.14	0.13
18	269592	313954	339066	**1.16**	**1.26**	0.15	0.13	0.12
19	269600	346727	272159	**1.29**	**1.01**	0.14	0.11	0.14
20	269600	282684	272160	**1.05**	**1.01**	0.14	0.13	0.13
21	272164	282689	272159	**1.04**	**1.00**	0.13	0.12	0.13
22	272163	296774	272158	**1.09**	**1.00**	0.12	0.11	0.12
23	272162	296782	272159	**1.09**	**1.00**	0.12	0.11	0.12
24	272162	296791	272158	**1.09**	**1.00**	0.11	0.10	0.11
25	276254	329561	276254	**1.19**	**1.00**	0.11	0.09	0.11
26	276255	397150	276258	**1.44**	**1.00**	0.10	0.07	0.10
27	262679	262680	262912	**1.00**	**1.00**	0.10	0.10	0.10

JPEG Encoder (13 Pages) : Summary of Temporal Partitioning Algorithms

Array Size	Execution Time			Improvement		Measured Array Activity		
	Exhaustive	Topological	Mincut	Topo/Exh	Min/Exh	Exhaustive	Topological	Mincut
4	1394432	1611008	1394176	**1.16**	**1.00**	0.45	0.39	0.45
5	1107712	1624320	1394176	**1.47**	**1.26**	0.46	0.31	0.36
6	1116645	1116672	1116672	**1.00**	**1.00**	0.38	0.38	0.38
7	1065366	1096704	1065434	**1.03**	**1.00**	0.34	0.33	0.34
8	1065364	1072896	1065412	**1.01**	**1.00**	0.30	0.30	0.30
9	866254	1072896	1065411	**1.24**	**1.23**	0.32	0.26	0.26
10	820408	1072896	1065402	**1.31**	**1.30**	0.31	0.24	0.24
11	820392	1072896	1065405	**1.31**	**1.30**	0.28	0.21	0.22
12	820397	832492	1065398	**1.01**	**1.30**	0.26	0.25	0.20
13	786609	786651	786665	**1.00**	**1.00**	0.25	0.25	0.25

JPEG Decoder (12 Pages) : Summary of Temporal Partitioning Algorithms

	Execution Time			Improvement		Measured Array Activity		
Array Size	Exhaustive	Topological	Mincut	Topo/Exh	Min/Exh	Exhaustive	Topological	Mincut
3	1784064	1784064	2052864	**1.00**	**1.15**	0.53	0.53	0.46
4	1105920	1533696	1533696	**1.39**	**1.39**	0.64	0.46	0.46
5	1103104	1103104	1763328	**1.00**	**1.60**	0.52	0.52	0.32
6	1063424	1063553	1063424	**1.00**	**1.00**	0.45	0.45	0.45
7	1054720	1054720	1054720	**1.00**	**1.00**	0.39	0.39	0.39
8	820736	820736	820736	**1.00**	**1.00**	0.43	0.43	0.43
9	820736	820736	820736	**1.00**	**1.00**	0.39	0.39	0.39
10	820736	820736	820736	**1.00**	**1.00**	0.35	0.35	0.35
11	820736	820736	820736	**1.00**	**1.00**	0.32	0.32	0.32
12	786688	786688	786688	**1.00**	**1.00**	0.30	0.30	0.30

B.3 Application Performance and Regimes of Operations

Application: Wavelet Encoder (30 pages) at CMB Size (L): 128 Kbits

	Offchip Memory BW 8 bits per cycle					Offchip Memory BW 4 bits per cycle				
CP	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}
3	5216132	**1.55**	3361264	56	139118	12148828	**1.58**	7667522	56	139101
4	4714690	**1.51**	3118522	56	152212	11264519	**1.58**	7139960	56	152202
5	4450734	**1.70**	2613368	64	165952	10957180	**1.69**	6468970	64	165937
6	2844115	**1.51**	1886421	79	241299	7449484	**1.52**	4891488	77	235570
7	2045417	**1.59**	1287266	95	250144	5653968	**1.86**	3039555	96	239953
8	2110170	**1.89**	1113940	97	191532	5432235	**2.22**	2451374	95	180803
9	1822222	**2.01**	906200	150	199424	4411124	**2.21**	1993188	153	179936
10	764194	**1.39**	548422	1989	223317	1721746	**2.46**	698994	505	0
11	806596	**1.74**	463235	1989	129525	1801941	**3.63**	496646	995	0
12	700322	**1.80**	389993	195	71744	1374081	**3.52**	390414	125	0
13	513147	**1.50**	341763	170	0	568007	**1.66**	341763	170	0
14	334641	**1.22**	275114	331	0	359655	**1.31**	275112	331	0
15	276170	**1.01**	272498	331	0	278754	**1.02**	272498	331	0
16	273362	**1.00**	272991	1701	0	273363	**1.00**	272994	1701	0
17	272510	**1.00**	272793	8160	0	272512	**1.00**	272793	8160	0
18	272484	**1.00**	272481	8119	0	272478	**1.00**	272485	8119	0
19	272229	**1.00**	272494	8119	0	272229	**1.00**	272496	8119	0
20	271951	**1.00**	272500	3978	0	271951	**1.00**	272500	3978	0
21	271966	**1.00**	272764	3978	0	271964	**1.00**	272766	3978	0
22	271967	**1.00**	272514	3978	0	271969	**1.00**	272514	3978	0
23	268152	**1.00**	268390	3855	0	268150	**1.00**	268392	7709	0
24	268161	**1.00**	268178	1401	0	268161	**1.00**	268175	1401	0
25	268137	**1.00**	268139	862	0	268138	**1.00**	267876	680	0
26	267830	**1.00**	267834	806	0	267828	**1.00**	267830	806	0
27	267374	**1.00**	267374	2657	0	267376	**1.00**	267374	1009	0
28	265672	**1.00**	265662	496	0	265670	**1.00**	265662	463	0
29	265674	**1.00**	265672	727	0	265673	**1.00**	265670	727	0
30	263305	**1.00**	263308	1	0	263305	**1.00**	263308	1	0

Application: Wavelet Encoder (30 pages) at CMB Size (L): 256 Kbits

	Offchip Memory BW 8 bits per cycle					Offchip Memory BW 4 bits per cycle				
CP	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}
3	2267062	**1.34**	1692538	113	107986	4795857	**1.43**	3357634	113	107822
4	2153736	**1.33**	1615710	113	107224	4492198	**1.47**	3053244	113	107047
5	2009326	**1.38**	1455426	130	114178	4437910	**1.62**	2733989	130	113969
6	1238105	**1.50**	823543	194	133935	2637761	**1.87**	1410530	46	0
7	1017004	**1.78**	572392	129	0	2151584	**3.76**	572392	129	0
8	981699	**1.82**	538218	194	31974	1680184	**3.07**	548036	150	0
9	773102	**1.61**	481138	240	0	884999	**1.84**	481140	237	0
10	387984	**1.33**	291768	4051	0	412150	**1.41**	291770	4051	0
11	389162	**1.34**	291498	4051	0	400499	**1.37**	291498	4051	0
12	392106	**1.37**	287174	386	0	431400	**1.50**	287174	387	0
13	358524	**1.25**	286085	335	0	361836	**1.26**	286081	335	0
14	275708	**1.00**	275114	331	0	275708	**1.00**	275112	331	0
15	273379	**1.00**	272498	331	0	273379	**1.00**	272498	331	0
16	273362	**1.00**	272504	2061	0	273363	**1.00**	272500	2061	0
17	272510	**1.00**	272793	8182	0	272512	**1.00**	272793	7955	0
18	272484	**1.00**	272481	7939	0	272478	**1.00**	272485	7955	0
19	272229	**1.00**	272494	8101	0	272229	**1.00**	272496	14105	0
20	271951	**1.00**	272500	3970	0	271951	**1.00**	272500	4019	0
21	271966	**1.00**	272764	3970	0	271964	**1.00**	272766	4019	0
22	271967	**1.00**	272514	3970	0	271969	**1.00**	272514	4019	0
23	268154	**1.00**	268390	3889	0	268158	**1.00**	268392	13285	0
24	268161	**1.00**	268178	1809	0	268161	**1.00**	268175	1809	0
25	268137	**1.00**	268139	872	0	268138	**1.00**	268142	869	0
26	267834	**1.00**	267834	2302	0	267828	**1.00**	267830	813	0
27	267374	**1.00**	267374	2657	0	267376	**1.00**	267374	1024	0
28	265672	**1.00**	265662	521	0	265670	**1.00**	265662	463	0
29	265676	**1.00**	265672	716	0	265673	**1.00**	265670	727	0
30	263308	**1.00**	263308	1	0	263305	**1.00**	263308	1	0

Application: Wavelet Encoder (30 pages) at CMB Size (L): 512 Kbits

	Offchip Memory BW 8 bits per cycle					Offchip Memory BW 4 bits per cycle				
CP	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}
3	1211836	**1.20**	1012716	182	43213	1530190	**1.49**	1025704	116	0
4	1084009	**1.30**	832682	178	20305	1412382	**1.55**	910469	120	0
5	1030598	**1.33**	776253	198	14565	1316788	**1.53**	858749	198	14565
6	646156	**1.60**	403171	389	42278	746448	**1.80**	415114	389	43814
7	515809	**1.45**	356634	391	16704	609932	**1.70**	359656	391	16704
8	566776	**1.68**	338086	389	16128	636002	**1.85**	343034	389	14848
9	508438	**1.61**	316004	613	18752	530000	**1.65**	322134	613	17216
10	291648	**1.00**	292084	4071	0	291648	**1.00**	291658	4951	0
11	291408	**1.00**	290994	4071	0	291406	**1.00**	291264	5556	0
12	286228	**1.00**	286926	617	0	286234	**1.00**	286729	601	0
13	286996	**1.00**	285600	457	0	286996	**1.00**	285632	457	0
14	277298	**1.01**	275110	330	0	277298	**1.01**	275112	330	0
15	273379	**1.00**	272518	330	0	273379	**1.00**	272498	330	0
16	273362	**1.00**	272468	2010	0	273363	**1.00**	272500	2051	0
17	272512	**1.00**	272788	8037	0	272512	**1.00**	272793	8037	0
18	272484	**1.00**	272483	8037	0	272486	**1.00**	272485	8037	0
19	272234	**1.00**	272496	21321	0	272237	**1.00**	272496	21321	0
20	271951	**1.00**	272505	4593	0	271951	**1.00**	272501	4593	0
21	271966	**1.00**	272768	11973	0	271964	**1.00**	272774	11973	0
22	271967	**1.00**	272516	7053	0	271969	**1.00**	272522	7053	0
23	268154	**1.00**	268390	3937	0	268158	**1.00**	268392	13285	0
24	268161	**1.00**	268175	1816	0	268161	**1.00**	268175	1816	0
25	268137	**1.00**	268142	865	0	268138	**1.00**	268142	865	0
26	267834	**1.00**	267828	4321	0	267828	**1.00**	267830	811	0
27	267374	**1.00**	267373	1783	0	267376	**1.00**	267376	10099	0
28	265672	**1.00**	265666	525	0	265670	**1.00**	265668	525	0
29	265678	**1.00**	265670	787	0	265673	**1.00**	265670	787	0
30	263308	**1.00**	263308	1	0	263305	**1.00**	263308	1	0

Application: Wavelet Decoder (27 pages) at CMB Size (L): 512 Kbits

	Offchip Memory BW 8 bits per cycle					Offchip Memory BW 4 bits per cycle				
CP	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}
6	1148408	**1.00**	1148372	607	217932	2403425	**1.03**	2342876	607	217932
7	980786	**1.02**	960510	1081	270949	2153930	**1.08**	1988611	777	255306
8	1487126	**1.20**	1240614	389	284083	3958462	**1.45**	2728318	389	284083
9	2206230	**1.19**	1858326	209	268625	5278564	**1.32**	3983855	257	263435
10	705034	**1.00**	704838	745	250816	1490498	**1.07**	1392662	745	250816
11	1227581	**1.71**	717373	592	281904	2788108	**2.19**	1274650	100	47981
12	833456	**1.25**	668672	355	152101	1910062	**2.30**	829864	115	0
13	746804	**1.38**	540876	1373	38229	1159692	**1.63**	709996	1373	38229
14	306946	**1.00**	306770	4506	0	326274	**1.06**	306770	4645	0
15	311214	**1.02**	306430	5101	0	325758	**1.06**	306442	4043	0
16	310916	**1.03**	301962	4081	0	321499	**1.06**	301962	4043	0
17	311728	**1.03**	301272	6259	0	311854	**1.04**	301263	5634	0
18	301865	**1.03**	292126	4537	0	301865	**1.03**	292138	4172	0
19	301241	**1.01**	296991	6133	0	301936	**1.02**	296989	7440	0
20	291917	**1.00**	291921	7141	0	291923	**1.00**	291923	7139	0
21	291896	**1.00**	291897	14536	0	291896	**1.00**	291896	14535	0
22	305840	**1.00**	305846	8401	0	305840	**1.00**	305840	8429	0
23	305316	**1.00**	305314	16661	0	305318	**1.00**	305314	16599	0
24	305294	**1.00**	305295	32662	0	305295	**1.00**	305295	32681	0
25	347589	**1.00**	347586	34201	0	347586	**1.00**	347586	33197	0
26	422409	**1.00**	422391	45801	0	422387	**1.00**	422387	45114	0
27	262772	**1.00**	263090	1	0	262776	**1.00**	262776	1	0

Application: JPEG Decoder (12 pages) at CMB Size (L): 128 Kbits

	Offchip Memory BW 8 bits per cycle					Offchip Memory BW 4 bits per cycle				
CP	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}
3	2601363	**1.11**	2344879	1016	0	2601364	**1.11**	2344978	1016	0
4	2284720	**1.04**	2198540	1021	0	2284721	**1.04**	2198535	1021	0
5	1909577	**1.07**	1781305	996	0	1909578	**1.07**	1781218	996	0
6	1433361	**1.00**	1433547	8119	0	1433413	**1.00**	1433610	8119	0
7	1196440	**1.00**	1195987	8119	0	1196437	**1.00**	1196276	8119	0
8	849529	**1.00**	849427	8119	0	849530	**1.00**	849432	8119	0
9	848023	**1.00**	847877	7955	0	848028	**1.00**	847876	7955	0
10	848109	**1.00**	847853	8119	0	848112	**1.00**	847856	8119	0
11	838599	**1.00**	838461	10671	0	838598	**1.00**	838465	10671	0

Application: JPEG Decoder (12 pages) at CMB Size (L): 256 Kbits

	Offchip Memory BW 8 bits per cycle					Offchip Memory BW 4 bits per cycle				
CP	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}
3	2156427	**1.02**	2106881	2031	0	2110341	**1.00**	2106876	2031	0
4	1867999	**1.02**	1823267	2021	0	1890064	**1.04**	1823260	2021	0
5	1535596	**1.00**	1535269	2031	0	1535411	**1.00**	1535270	2031	0
6	1228961	**1.00**	1228986	16155	0	1254282	**1.02**	1228994	16155	0
7	1118088	**1.00**	1117881	16155	0	1127598	**1.01**	1117876	16155	0
8	829698	**1.00**	829934	16155	0	839568	**1.01**	829931	16155	0
9	829314	**1.00**	829207	16237	0	839502	**1.01**	829204	16237	0
10	829264	**1.00**	829373	16155	0	838510	**1.01**	829368	16155	0
11	829135	**1.00**	829135	20384	0	829134	**1.00**	829134	20384	0

Application: JPEG Decoder (12 pages) at CMB Size (L): 512 Kbits

	Offchip Memory BW 8 bits per cycle					Offchip Memory BW 4 bits per cycle				
CP	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}
3	1945803	**1.01**	1922899	4081	0	1945804	**1.01**	1922900	4081	0
4	1670615	**1.01**	1648759	4081	0	1670616	**1.01**	1648760	4081	0
5	1269233	**1.02**	1246753	4081	0	1269234	**1.02**	1246754	4081	0
6	1152284	**1.00**	1152195	30013	0	1152288	**1.00**	1152210	30013	0
7	1078961	**1.00**	1078761	32637	0	1078964	**1.00**	1078764	32637	0
8	819862	**1.00**	819862	32637	0	819865	**1.00**	819865	32637	0
9	819847	**1.00**	819847	32637	0	819845	**1.00**	819845	32637	0
10	819830	**1.00**	819830	32637	0	819833	**1.00**	819833	32637	0
11	819815	**1.00**	819815	32701	0	819814	**1.00**	819814	32701	0
12	786631	**1.00**	786631	1	0	786628	**1.00**	786628	1	0
13	786631	**1.00**	786631	1	0	786630	**1.00**	786630	1	0
14	786632	**1.00**	786632	1	0	786636	**1.00**	786636	1	0
15	786634	**1.00**	786634	1	0	786636	**1.00**	786636	1	0

Application: JPEG Encoder (13 pages) at CMB Size (L): 128 Kbits

	Offchip Memory BW 8 bits per cycle					Offchip Memory BW 4 bits per cycle				
CP	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}
2	14890717	**2.68**	5550276	128	34069	31532422	**5.12**	6155608	82	0
3	12115886	**2.86**	4232586	110	5693	25047772	**5.43**	4615362	118	5686
4	4677392	**1.67**	2808931	128	0	5836786	**2.08**	2809018	128	0
5	2088453	**1.00**	2083566	1377	0	2095592	**1.01**	2083515	1377	0
6	1288733	**1.03**	1253231	538	0	1288736	**1.03**	1253234	511	0
7	1348356	**1.08**	1253717	529	0	1348355	**1.08**	1253745	544	0
8	1308353	**1.04**	1253745	544	0	1308356	**1.04**	1253766	538	0
9	1308049	**1.04**	1253401	544	0	1308072	**1.04**	1253394	544	0
10	849891	**1.00**	849883	14597	0	849896	**1.00**	849882	14597	0
11	786495	**1.00**	786496	1	0	786492	**1.00**	786492	1	0

Application: JPEG Encoder (13 pages) at CMB Size (L): 256 Kbits

	Offchip Memory BW 8 bits per cycle					Offchip Memory BW 4 bits per cycle				
CP	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}
2	6981905	**2.15**	3243353	236	5074	11799863	**3.54**	3331760	216	0
3	5811477	**1.84**	3158453	213	5524	11090493	**3.42**	3244544	199	0
4	2872149	**1.33**	2155437	234	0	3007352	**1.38**	2173509	248	0
5	1587329	**1.02**	1559875	2657	0	1592267	**1.02**	1559874	2657	0
6	1164052	**1.01**	1148911	1021	0	1163771	**1.01**	1148910	1021	0
7	1203730	**1.04**	1160557	1016	0	1203727	**1.04**	1160560	1016	0
8	1197415	**1.03**	1157151	811	0	1197414	**1.03**	1157152	811	0
9	1189552	**1.04**	1146979	1021	0	1189554	**1.04**	1146980	1021	0
10	840694	**1.00**	840686	21977	0	840696	**1.00**	840691	21977	0
11	786495	**1.00**	786492	1	0	786496	**1.00**	786492	1	0

Application: JPEG Encoder (13 pages) at CMB Size (L): 512 Kbits

	Offchip Memory BW 8 bits per cycle					Offchip Memory BW 4 bits per cycle				
CP	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}	MS_{fixed}	$fixed/var$	MS_{var}	Q_{simopt}	B_{spill}
2	4039039	**1.66**	2435586	445	0	5862290	**2.41**	2435585	445	0
3	3508977	**1.50**	2338037	451	10151	4586612	**1.93**	2378752	388	0
4	2027431	**1.23**	1653857	511	0	2035353	**1.23**	1653832	511	0
5	1346633	**1.00**	1346609	5413	0	1346632	**1.00**	1346610	5413	0
6	1101715	**1.00**	1101068	2047	0	1101718	**1.00**	1101072	2102	0
7	1117291	**1.01**	1105008	2113	0	1117286	**1.01**	1105012	2058	0
8	1112879	**1.01**	1100595	2091	0	1112878	**1.01**	1100594	2047	0
9	1112567	**1.01**	1100285	2047	0	1112570	**1.01**	1100286	2102	0
10	831507	**1.00**	831499	44281	0	831505	**1.00**	831496	44281	0
11	786492	**1.00**	786492	1	0	786492	**1.00**	786492	1	0

Bibliography

[BML96] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[Buc93] Joseph Tobin Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token flow Model*. PhD thesis, UC Berkeley, 1993.

[Buc94] Joseph T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *Conference on Signals, Systems, and Computers*, November 1 1994.

[CCH$^+$00] Eylon Caspi, Michael Chu, Randy Huang, Nicholas Weaver, Joseph Yeh, John Wawrzynek, and André DeHon. Stream computations organized for reconfigurable execution (score): Extended abstract. In *Conference on Field Programmable Logic and Applications (FPL '2000)*, pages 605–614. Springer-Verlag, August 28-30 2000.

[Chu00] Michael Monkang Chu. Dynamic runtime scheduler support for score. Master's thesis, UC Berkeley, 2000.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press/McGraw Hill, 1990.

[DeH96] André DeHon. *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis, MIT, 545 Technology Sq., Cambridge, MA 02139, September 1996.

[FPR96] H. Franke, P. Pattnaik, and L. Rudolph. Gang scheduling for highly efficient, distributed multiprocessor systems. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, page 4. IEEE Computer Society, 1996.

[GDWL92] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability*. W.H. Freeman, New York, 1979.

[GLL99] Alain Girault, Bilung Lee, , and Edward A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, 18(6), June 1999.

[Ha92] Soonhoi Ha. *Compile-Time Scheduling of Dataflow Program graphs with Dynamic Constructs*. PhD thesis, UC Berkeley, 1992.

[HL91] S. Ha and E. A. Lee. Quasi-static scheduling for multiprocessor dsp. In *IEEE International Symposium on Circuits and Systems, Singapore. Conference on Signals, Systems, and Computers*, June 1991.

[HL97] Soonhoi Ha and Eward A. Lee. Compile-time scheduling of dynamic constructs in dataflow program graphs. *IEEE Transactions on Computers*, 46(7), July 1997.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In *IFIP Congress 74*, pages 471–475, 1974.

[Lee91] Edward Lee. *Advanced Topics in Data-Flow Computing*, chapter Static Scheduling of Data-Flow Programs for DSP, pages 501 – 527. Prentice-Hall, Inc., 1991.

[LW98] Huiqun Liu and D. F. Wong. Network-flow-based multiway partitioning with area and pin constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(1):50 – 59, January 1998.

[MCH$^+$02] Yury Markovskiy, Eylon Caspi, Randy Huang, Joseph Yeh, Michael Chu, André DeHon, and John Wawrzynek. Analysis of quasi-static scheduling techniques in a virtualized reconfigurable machine. In *Tenth International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 196–205, February 24-26 2002.

[MKF$^+$01] Rafael Maestre, Fadi J. Kurdahi, Milagros Fernández, Roman Hermida, Nader Bagherzadeh, and Hartej Singh. A framework for reconfigurable computing: task scheduling and context management. *IEEE*

Transactions on Very Large Scale Integrated Systems, 9(6):858–873, 2001.

[Par95] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, UC Berkeley, 1995.

[RPZM93] S. Ritz, M. Pankert, V. Zivojinovic, and H. Meyr. Optimum vectorization of scalable synchronous dataflow graphs. In *International Conference on Application-Specific Array Processors*, pages 285–296, 1993.