

Analysis of Quasi-Static Scheduling Techniques in a Virtualized Reconfigurable Machine

Yury Markovskiy, Eylon Caspi, Randy Huang,
Joseph Yeh, Michael Chu, John Wawrzynek
University of California, Berkeley

{yurym, eylon, rhuang, jyeh, mmchu, johnw}@cs.berkeley.edu

André DeHon
California Institute of Technology
andre@acm.org

ABSTRACT

The SCORE compute model uses fixed-size, virtual compute and memory pages connected by stream links to capture the definition of a computation abstracted from the detailed size of the physical hardware. When the number of physical compute pages is smaller than the number of virtual compute pages in the abstract computation graph, the design is time-multiplexed onto the available physical hardware. A key component of this strategy is an automatic scheduler that selects the temporal sequencing of virtual resources onto the physical device. We describe a quasi-static scheduling strategy that retains the full semantic power of the *dynamic* SCORE flow graph while taking advantage of static scheduling techniques at program load time to hoist most of the computational work out of the inner scheduling loops. This strategy reduces online scheduling work per reconfiguration epoch by an order of magnitude. In addition, a more global perspective available from offline-scheduling improves schedule quality, resulting in a net reduction of total execution time by 46–81%.

1. INTRODUCTION

Reconfigurable computing devices such as FPGAs have demonstrated 10x-100x gains in performance and functional density over microprocessors for a variety of applications [6], yet their popular use is limited to application-specific domains or serving as ASIC replacements. These uses ignore the devices' programmability and limits their applicability to only a few areas in computing. Whereas microprocessor architectures have traditionally enjoyed software compatibility and automatic performance scaling across device generations, applications developed for current reconfigurable

devices are usually limited to one target and require manual adaptation to take advantage of new, larger and faster targets.

SCORE [4] strives to eliminate existing barriers to widespread efficient exploitation of reconfigurable devices by introducing a compute model based on paged virtual hardware (similar to virtual memory). The paged model provides a framework for device size abstraction, automatic run-time 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* (FSMs) that communicate tokens¹ through streams (FIFO channels) 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 microarchitecture. Our current target hardware is a microprocessor/reconfigurable array hybrid architecture shown in Figure 1. The *array* is partitioned into *fixed-size* compute pages (CPs) and configurable memory blocks (CMBs). The run-time scheduler time-multiplexes compute pages on physical CPs and manages buffer allocation in CMBs to provide the user with the illusion of unbounded hardware.

The separation between the compiler and the run-time scheduler is similar in purpose to the traditional separation between the function of a compiler and an operating system. The compiler is responsible for transforming a high-level representation of the computation into a binary form suited for configuring individual hardware resources on the array. The scheduler is responsible for binding and managing those resources at execution time. This separation allows delaying certain mapping decisions until execution time and adapting those decisions to the specific hardware configuration being used. Adapting to different device sizes is critical to application longevity, enabling automatic performance scaling on next generation hardware.

SCORE computation graphs permit data-dependent token consumption/emission, resulting in dynamic flow rates and making it impossible to place static bounds on memory

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FPGA'02, February 24-26, 2002, Monterey, California, USA.
Copyright 2002 ACM 1-58113-452-5/02/0002 ...\$5.00.

¹A unit of communication with respect to stream operations.

usage. The latter *requires* a scheduler to manage stream buffer sizes at run-time to guarantee execution correctness. More restrictive models of computation such as synchronous (SDF) [1], boolean controlled (BDF) and integer controlled data-flow (IDF) [2, 3] define necessary conditions for graph consistency and bounds on buffering requirements, and therefore allow a simpler, purely static scheduling methodology.

The scheduling problem discussed in this paper is an NP-hard optimization problem with multiple, simultaneous independent constraints on memory, communication bandwidth and available resources. The literature offers a wealth of efficient and near-optimal scheduling solutions for restricted data-flow compute models (*e.g.* synchronous data-flow[1]). We demonstrate that SCORE graphs can be scheduled efficiently while retaining the full semantic power and expressiveness of our stronger compute model.

The main objective of this work was to avoid the high run-time overhead of a fully dynamic scheduler without restricting data-dependent data-flow in SCORE. In addition to drastically reducing the run-time overhead (computational workload) of scheduling, the quasi-static scheduler produces better quality schedules than the dynamic scheduler. The net result is an average factor of 3 reduction in the total run-time for a range of applications.

This paper is organized as follows. In Section 2, we examine a fully dynamic scheduler. Section 3 presents a space of available scheduling solutions and classifies several feasible implementations in a taxonomy. Sections 4 and 5 analyze our quasi-static scheduler and its implementation methodology. Section 6 presents results.

2. FULLY DYNAMIC SCHEDULER

Because few restrictions exist on token flow behavior in SCORE compute graph nodes, a fully dynamic run-time scheduler [5] was a natural choice for our initial system implementation. The scheduler, designed to handle large data-dependent variations in page token consumption and emission rates, makes decisions driven largely by token availability at page inputs. A dynamic scheduler can continuously monitor active computation progress on the array and adapt the schedule to optimally match the observed application data-flow patterns. The resulting schedule quality depends heavily on the temporal *granularity* (frequency) 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.

2.1 Operation and Structure

Given an application graph and resource constraints (*e.g.* number of CPs and CMBs), our dynamic scheduler attempts to time-multiplex virtual graph nodes on physical resources to minimize application total execution time (*makespan*).

The scheduler manages execution of application graphs on the array by issuing reconfiguration and memory transfer commands to the array controller. During execution, the scheduler is invoked once for each *fixed time-slice* (250,000 cycles in our implementation), as shown in Figure 2. At each invocation, the scheduler queries the array state, decides to evict and/or schedule particular nodes, reconfigures the array, and resumes array execution.

Before examining each individual scheduling module in detail, let us look at two simple execution scenarios and the

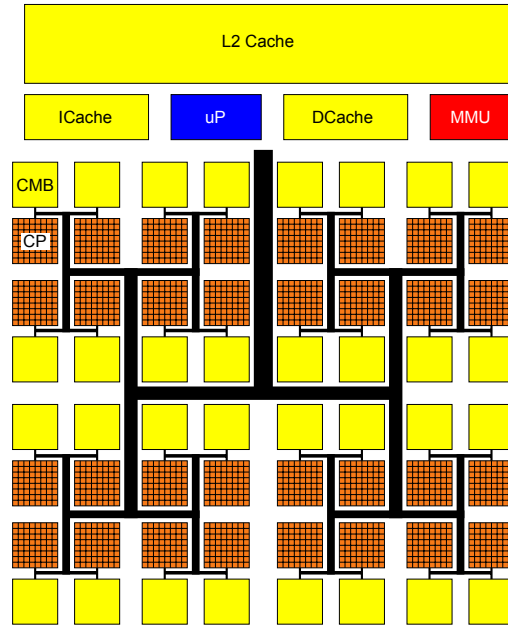


Figure 1: Hypothetical, single-chip SCORE system.

scheduler’s role in them:

- *Small Design, Large Array* is the the simplest scenario, where the entire design fits on the array. The role of the scheduler is reduced to mapping a compute page graph to selected physical resources once.
- *Large Design, Small Array* is a common scenario where a design requires more resources than available, and the scheduler time-multiplexes graph nodes onto physical resources. The scheduler is invoked once per time-slice; 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 *bufferlock*, an infrequently occurring condition resulting from implementing a SCORE application, which assumes unbounded stream buffers, on a physical array with limited resources (see [12] for a formal treatment). Some applications may require streams to buffer a greater number of tokens than the array stream hardware permits, and failure to provide this buffering capacity would result in deadlock. The scheduler monitors the progress made by a design and checks for potential deadlock conditions. If any are detected and the cause is determined to be limited buffering capacity of a stream, the scheduler intervenes to resolve the *bufferlock*. The offending stream is “cut,” and a memory block configured as a FIFO is inserted to provide additional buffering. Array execution then resumes.

Complete treatment of the dynamic scheduler operation can be found in [5]. Here we shall briefly discuss the module flow and critical data structures. 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 that satisfy precedence constraints (their predecessors have been/are scheduled) 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

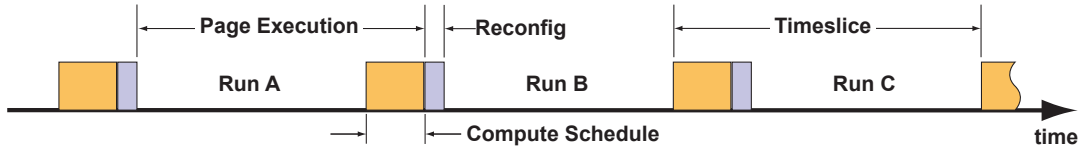


Figure 2: Application Execution Timeline.

output space. Alternatively the algorithm can be thought of as a greedy, breadth-first packing of nodes onto the array.

Figure 5 demonstrates the relationship between modules of the dynamic scheduler. The scheduler is invoked in each time-slice to perform the following sequence of operations:

- *Query Array State* obtains execution statistics from the array hardware, updates corresponding scheduler data structures, and 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 handled as described above, and deadlocked processes are killed.
- *Schedule* computes available array resources after removing the nodes that terminated or exhibited low firing activity. It then attempts to pack the array with the “frontier” priority list nodes that are expected to make progress if scheduled. The scheduler inserts special “stitch” segments to buffer the contents of streams crossing temporal partitions. This module’s output is a page subgraph, guaranteed to fit on the array, to be scheduled in the following time-slice.
- *Resource Allocation* assigns the subgraph nodes to specific physical compute pages and memory blocks.
- *Reconfigure* issues a sequence of commands to load the subset of nodes selected by the previous modules and to resume execution.

2.2 Analysis of Performance

Figure 3 shows performance results 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, and the horizontal axis shows the array size (in compute page/configurable memory block pairs). Performance was measured on a cycle-level array simulator. Two curves are shown on the diagram—the execution times on a simulated real and a simulated idealized system; the latter does not include the run-time computational overhead of the scheduler.

Both curves exhibit expected performance scaling behavior. In general, more hardware makes for 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; all non-essential components were eliminated, and remaining code was redesigned to improve memory allocation and layout of data-structures in an attempt to reduce run-time overhead while maintaining schedule quality. However, the average run-time overhead observed from the results on Figure 3 is still about 36% of the total execution time. Similar high run-time overhead is observed with other applications, including JPEG and wavelet codecs. Having no basis for comparison, little can be concluded

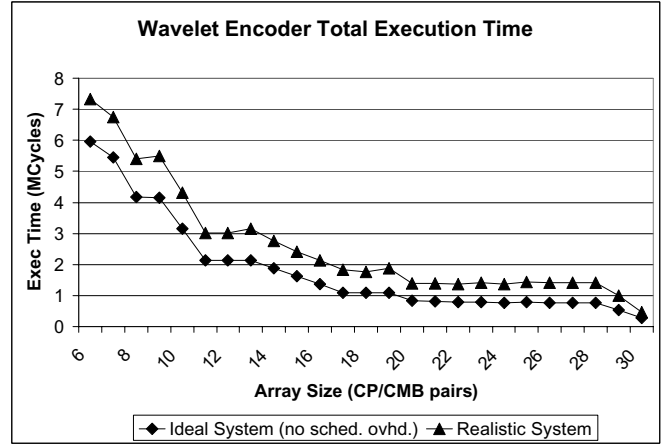


Figure 3: Wavelet Encoder (30 pages): Total execution time with the fully dynamic scheduler.

about scheduling quality even in an idealized system, but clearly more complex algorithms than the one implemented would be required to further improve application performance. With 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 provides somewhat acceptable results for applications with very large (100,000s of cycles) or unlimited total execution time, even with its average run-time scheduling overhead of 119,000 cycles per time-slice. However, high scheduling overhead makes a SCORE implementation inefficient in the following cases:

- The performance of applications with short total execution time (*e.g.* below 100,000 cycles) 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 delays (tokens) in the feedback loop. If this number of delays is small compared to the reconfiguration and scheduling time, then reconfiguration and scheduling overhead will dominate useful computing time. This is a similar phenomenon to virtual memory thrashing that occurs when the working set does not fit into available physical memory.

Since our micro-architectural design exhibits array reconfiguration time on the order of only 10,000 cycles, dynamic scheduling becomes the primary bottleneck preventing efficient execution of the aforementioned applications.

3. SPACE OF SCHEDULING SOLUTIONS

The problem of low run-time overhead scheduling of data-flow graphs is not new and has been solved for some restricted data-flow models, such as synchronous data-flow (SDF) [1]. Under SDF, for example, the data token input and output rates of individual nodes are static, permitting a

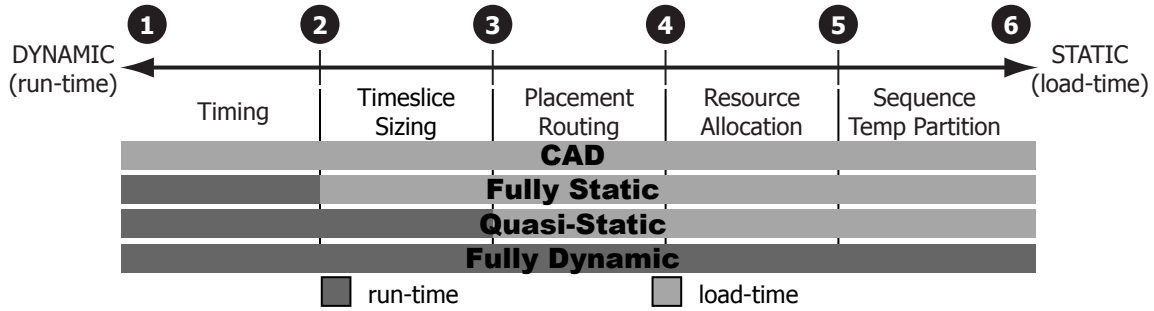


Figure 4: Space of scheduling solutions, which includes (1) FPGA CAD, (2) a fully static (e.g. SDF), and (6) a fully dynamic scheduler (e.g. Section 2). Quasi-Static scheduler (3) is discussed in Section 4.

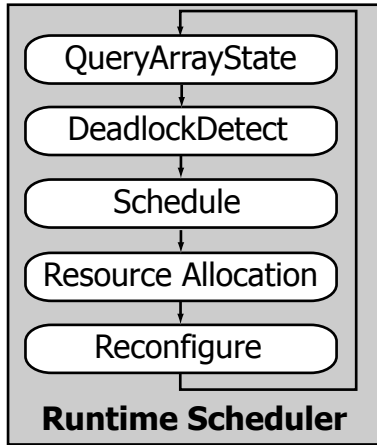


Figure 5: Fully Dynamic Scheduler: module flow in the critical loop.

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, completely avoiding run-time overhead. However, a dynamic scheduler makes no *a priori* assumptions about application behavior and operates purely by observing instantaneous computation progress. While a dynamic scheduler allows flexibility in handling dynamic data-driven behavior and run-time graph construction, it comes at the cost of high run-time overhead.

Notice that in both extremes of fully dynamic and fully static scheduling, the same basic set of operations is performed: computing node firing sequence and timing, and allocating 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 in order to understand the opportunities that exist for low overhead/high quality scheduling.

In [10] Lee forms a taxonomy of scheduling solutions and explores the space between fully static and fully dynamic approaches. Lee attempts to find a compromise between a low overhead static and a high overhead dynamic scheduling for applications with data-dependent data-flow. Lee demonstrates efficient solutions, combinations of static and dynamic scheduling techniques required to handle BDF and IDF. For those data-flow models, 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 [9, 7, 8].

We wish to build and expand on the taxonomy in [10] by identifying its analogue for our system. For SCORE we define five specific inter-dependent steps that must be performed to schedule a design on a reconfigurable array (see Figure 4). One way to represent a spectrum of run-time resource management solutions for our system 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 statically. For example, a point marked as 3 represents a scheduling solution where steps to the left (*Timing* and *Timeslice Sizing*) are performed dynamically (at run-time) and steps to the right (*Place/Route*, *Resource Alloc*, and *Sequence/Temp Partition*) are performed statically (at application load/install time).

Figure 4 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, we shall use 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 subgraphs. A schedulable subgraph is one that “fits” on the array; each virtual page requires a physical CP; each user memory segment—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) creates an NP-hard problem. Optimizations include minimizing buffer requirements, maximizing hardware utilization, avoiding cuts in graph cycles to prevent thrashing, and increasing temporal locality of intermediate data.
- *Resource Allocation* maps the schedulable subgraphs down to actual 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 no assumptions were previously made about the array’s routing structure nor

about its physical layout. Should routing or placement fail, the scheduler may return to *Temporal Partitioning* and repeat that step with tighter constraints.

- *Timeslice Sizing* (Subgraph Execution Time) 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 ability to stall compute pages (CPs) and configurable memory blocks (CMBs). Similarly, microprocessors use score-boarding to tolerate variable delay in memory and arithmetic operations. 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 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. Some steps (*e.g.* Timing) may be implemented efficiently in array hardware, thus obviating direct involvement of the run-time software scheduler. Figure 4 marks 1 through 6 as clear places where cuts, that divide the space into dynamic and static sub-spaces, can turn into implementations. A complete discussion of the proper balance of efficiency, functionality and flexibility is beyond of the scope of this paper. Below we summarize issues driving the selection of specific solutions based on the underlying implementation requirements.

- **Run-Time overhead** is the most obvious. The scheduler at 1 incurs no run-time overhead, but the overhead gradually increases as the scheduler implementation performs more steps at run-time. Run-Time overhead depends heavily on the granularity of scheduler involvement in application execution and on scheduling algorithm complexity.
- **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 (*e.g.* in SDF, close-to-optimal schedules can be constructed statically). As a scheduler performs more steps at run-time (*i.e.* moving from 2 toward 6), application behavior can be monitored, not only predicted. For every scheduler in between 2 and 6 both *accuracy* of predicted and *currency* of observed information determine scheduling quality. There is a tradeoff 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** may be supported in solutions that perform more steps dynamically. SCORE permits dynamic instantiation of graph nodes and cre-

ation of subgraphs at run-time, allowing a computation to be composed dynamically and sharing the array between applications. These features require *Resource Allocation* to map virtual to physical resources at run-time.

Every solution in the space outlined above can implement the complete semantics of SCORE, but only a subset will perform efficiently. Independent of the actual choice, a great variety of control-dominated computations can be expressed without restrictions (compression, sorting, selection and many others), allowing efficient exploitation of the powerful semantics of SCORE in a low overhead scheduling implementation.

4. QUASI-STATIC SCHEDULER

Having analyzed in the previous section several points in the space of scheduling solutions, here we describe a viable and efficient implementation that corresponds to point 3 on Figure 4. We call the implementation a *quasi-static scheduler* because it adapts to slowly varying run-time characteristics (graph composition and firing rates) by recomputing a high quality, static schedule only when the computation graph changes substantially.

4.1 Basic Implementation Principles

The quasi-static scheduler expands on existing work to map synchronous data-flow (SDF) programs to uni- and multi-processors [1]. An SDF program is a data-flow graph whose computational nodes (*actors*) communicate via *arcs* using the same streaming discipline as SCORE (an arc is a FIFO with logically unbounded capacity). Analyses and algorithms for SDF can be adapted for SCORE by equating a page with an actor. SDF is well established in the literature and has been successfully used to map signal processing algorithms to a variety of hardware platforms.

Scheduling SCORE for a hybrid reconfigurable architecture differs in two key ways from scheduling SDF for microprocessors. First, the hardware models have different execution costs, and hence require different optimization criteria for scheduling (discussed below). Second, SDF actors are restricted to having static input/output rates (*e.g.* an adder that repeatedly consumes two inputs to produce one output), whereas SCORE pages may have dynamic input/output rates. 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.

The reconfigurable architecture of SCORE has different execution costs than a microprocessor running SDF and hence requires different optimization approaches. First, we note that an SDF program for a uni-/multi-processor target is typically scheduled at compile time as a collection of single-threaded instruction sequences, one per microprocessor, with each sequence repeatedly evaluating a subset (sub-graph) of actors. SDF techniques tend to cluster communicating actors in the same microprocessor, running them time-multiplexed with memory-buffered communication. In the uni-processor case, this style is inevitable because there is only one processor; in the multi-processor case, it is because inter-processor communication primitives are more expensive than memory access, typically consuming 10s to 100s of cycles. Given that, 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 (several clock cycles), so frequent actor switching is acceptable. On reconfigurable hardware, however, switching to evaluate a different page is expensive, requiring saving and restoring a page context (hundreds to thousands of clock cycles). Consequently, a SCORE schedule prefers to reuse (re-evaluate) 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 prefers to schedule communicating actors concurrently on separate pages.

It is important to demonstrate that the quasi-static scheduler provides the same computational semantics as the dynamic scheduler. That is, for correct execution of any SCORE graph, time-multiplexed execution using physically bounded buffers must get 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 discipline 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 will give every page an opportunity to fire on each schedule iteration.
- The quasi-static scheduler will iterate through its schedule until all pages have completed. Hence, regardless of the order in the schedule, every page will be given an opportunity to consume all of its inputs and produce all of its results.
- As long as the array has not deadlocked, the virtual graph will make forward computational progress on every schedule iteration.
- Should bufferlock occur, the scheduler will expand 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, will not deadlock when time-multiplexed onto limited physical resources by the quasi-static scheduler², and the quasi-static schedule will produce the same functional results as the unbounded case.

4.2 General System Tool Flow

The quasi-static scheduler consists of a static schedule generator and a run-time reconfiguration engine, as shown in Figure 6. The modules of the quasi-static scheduler are similar to those of the dynamic scheduler (Figure 5), but they have been factored into two components so that certain tasks are performed less frequently than every time-slice. The inner loop of the scheduler, *i.e.* the work that incurs run-time overhead at every time slice, has thus been substantially reduced. The infrequent component is the schedule generator, which analyzes the virtual page graph plus profile information from previous application runs to produce a schedule in the form of a script of array

²A deadlock can occur only if application’s total buffering requirements exceed the physical system memory.

Array Resource	Time Step		
	1	2	3
CP0	A	B	C
CP1	D		J
CMB0	K[0:10]	K[0:10]	M[11:20]
CMB1	N[2:20]	O[40:45]	P[25:39]

Table 1: A static schedule 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 within CMBs.

reconfiguration commands. The frequent component (inner loop) is the reconfiguration engine, which oversees execution of the graph by issuing script commands to the hardware. Particular differences from the dynamic scheduler are as follows:

- *Query Array State* was drastically simplified to only detect and process “done” signals from nodes.
- *Deadlock* detection and resolution are not currently implemented, but will be added in the future. It is important to note that deadlock detection remains cheap under quasi-static scheduling, requiring page activity counters in hardware and minimal house-keeping in software. Resolving deadlock remains expensive and typically requires regenerating a schedule script.
- *Schedule* and *Resource Allocation* were moved to the static schedule generator.
- *Reconfigure* was replaced by a light-weight script execution engine.

By simplifying or eliminating required run-time components, the quasi-static scheduler incurs on average only *one tenth* of the per-time-slice run-time overhead of the fully dynamic implementation. More comprehensive results will be discussed in following sections.

4.3 Static Schedule Generator

The static schedule generator analyzes the virtual page graph and application execution profile to compute a reconfiguration script (Figure 6). We described the first two modules *Partition* and *Resource Allocation* in detail in Section 3. 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, as shown on Table 1. Each column represents a scheduling step (time-slice), while 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.

To simplify the job of the run-time reconfiguration engine, the schedule emitted by the static schedule generator is represented as a script of commands, instead of a table. Two types of commands are used: (1) control commands (*e.g.* wait for the next time-slice, execute N cycles, *etc.*) interpreted by the run-time script execution engine, and (2) reconfiguration commands (*e.g.* transfer configuration bitstream from CMB to CP, set CMB memory bounds registers, *etc.*) forwarded directly to the array.

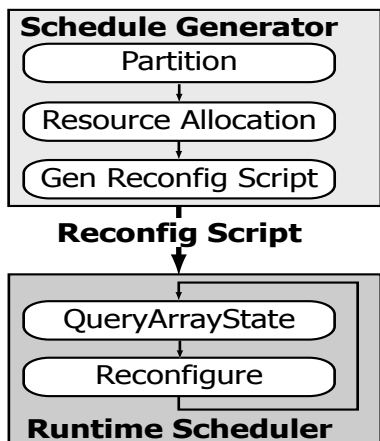


Figure 6: Quasi-Static Scheduler: module flow in static and run-time components.

4.4 Hardware Support

The key difference between a fully static scheduler (point 2 in Figure 4) and the quasi-static scheduler (point 3) is the time the *Timeslice Sizing* step is performed. The *Timeslice Sizing* step computes a time interval for each schedulable subgraph to be resident on the array. While a fully static scheduler must specify *a priori* precise periods of time to schedule each subgraph, our quasi-static implementation relies on special array hardware to detect stall conditions on the array. On detection of stall conditions, the run-time reconfiguration engine moves to the next scheduling step.

Stall conditions are those that impede any progress of the resident subgraph, such as an *empty* input or *full* output buffer. The reconfiguration script includes commands to allow CMBs and the global array controller to detect these conditions. When a stall condition occurs, the array controller issues an interrupt to invoke the run-time reconfiguration engine on the microprocessor. The engine then schedules the next subgraph in the static 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 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 fully dynamic scheduler uses a fixed time-slice for the *Timeslice Sizing* step and analyzes run-time page activity in *Sequence* and *Resource Allocation*. In contrast, the quasi-static scheduler with *stall detect* analyzes CP activity in *Timeslice Sizing* and uses predicted application behavior for other steps at load-time. The quasi-static scheduler uses *buffer sizing* to control subgraph execution time between scheduler actions, *i.e.* vary time-slice size (Section 5.2).

5. QUASI-STATIC SCHEDULING ALGORITHMS

5.1 Temporal Partitioning

Partition, the key component that determines scheduling quality, is the first step performed by the static schedule generator. It computes the execution sequence of nodes

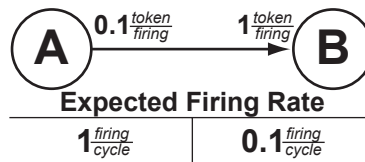


Figure 7: Rate mismatches between co-resident nodes lead to hardware underutilization.

and divides a virtual page compute graph into schedulable subgraphs to be time-multiplexed on the reconfigurable array.

The static scheduler uses graph topology and node token emission/consumption rates obtained through profiling to predict application behavior. While it is generally advantageous to make neighbor nodes co-resident, effectively creating pipelines of pages, an I/O token rate mismatch between the neighbors could result in a serious under-utilization of array hardware (discussed below). The static schedule generator attempts to improve performance by computing near-optimal graph partitioning to maximize array hardware utilization.

5.1.1 Performance Model

An I/O rate mismatch between co-resident adjacent nodes frequently leads to significant underutilization of array hardware. A simple example in Figure 7 shows two virtual pages, A and B, co-scheduled on a reconfigurable array with two CPs. Each page has intrinsic I/O rates, expressed in units of *tokens/firing*. Page A is a slow producer and limits the ability of page B to fire. A can fire at the rate of $1 \frac{\text{firing}}{\text{cycle}}$, whereas B only at $0.1 \frac{\text{firing}}{\text{cycle}}$ (*i.e.* B fires once every ten cycles when a token appears on its input). Let us assume that virtual node A is mapped to CP0 and B to CP1 on the array. Intrinsic I/O rates of each node allow CP0 to be utilized 100% of the time and CP1 only 10%, leading to average CP utilization of $(1 + 0.1)/2 = 0.55 = 55\%$.

Average CP utilization is a convenient measure of scheduling quality for a specific device size. The SCORE array simulator records the number of cycles each compute page (CP) fired during execution, and computes an *observed* average CP utilization:

$$\overline{U_{CP}} = \frac{\sum_{i=1}^N F_i}{MS * N} \quad (1)$$

where N is the array size (the number of CPs), F_i is the number of cycles that CP i fired, and MS is the makespan (total execution time) for an application. Note that, as expected, Equation 1 demonstrates an *inverse* relationship between average CP utilization $\overline{U_{CP}}$ and the total execution time in an ideal system (no scheduling overhead). The total execution time is denoted as MS .

While computing the *observed* average CP utilization permits simple comparison between partitioning algorithms, it does not readily identify what a partitioning algorithm should do to improve application performance. To fully understand optimizations and tradeoffs involved in temporal graph partitioning, a mathematical model based on SDF scheduling techniques was developed to define and quantify a relationship between average CP utilization and the total execution time in an ideal system (no scheduling overhead). Average CP utilization was also defined independently of

the execution time, based solely on individual nodes in each partition. Detailed discussion of this model is omitted for brevity, but the model is able to predict the *observed* average CP utilization with less than 5% average error in our experiments.

While application total execution time depends on the size of the input dataset, average CP utilization depends only on the schedule (*e.g.* individual co-resident pages, rate mismatches, etc). The *Partitioner* uses the mathematical model to estimate average CP utilization for a given candidate partition set and attempts 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 (resulting in the lowest total application execution time).

5.1.2 Temporal Partitioning Algorithms

Optimal graph partitioning under multiple independent simultaneous constraints (*e.g.* CP/CMB count) is an NP-hard optimization problem. To study the problem in detail, we have implemented two heuristic and one exhaustive search partitioner to be used as a reference.

Topological Partitioner uses a simple greedy packing algorithm with a precedence constrained traversal order. The algorithm starts by topologically sorting graph nodes, then iterates over the resultant node list forming schedulable subgraphs. Possessing only a very limited global view of the entire graph, this algorithm benefits from a special pre-clustering pass that decreases the number of streams crossing temporal partition boundary. The pre-clustering pass repeatedly merges any two clusters together as long as the resulting cluster I/O stream count is lower than the aggregate I/O stream count of the two separately. With topological sort implemented by a depth-first search, the complexity of this algorithm is $O(|E|)$, where $|E|$ is the number of graph edges.

Balanced N-way Mincut is a flow-based mincut partitioning algorithm, based on Wong’s temporal partitioning for FPGA circuits [11], and adapted to enforce precedence constraints. In its core, the algorithm performs a mincut on the graph and examines resulting partitions. The partition containing the nodes with the highest scheduling precedence is grown or shrunk until it fits on the array. To grow or shrink a partition, the algorithm augments/evicts a selected node and repeats the process until a satisfactory partition is obtained. The partitioner uses the average CP utilization model to select nodes to move between partitions. Its goal is to maximize predicted average CP utilization. This algorithm complexity is $O(|V||E|)$, where $|V|$ is the number of graph vertices and $|E|$ —edges.

Exhaustive Search partitioner examines every possible schedule to find the one with highest average CP utilization. This algorithm is our performance reference for evaluating the quality of the two heuristic partitioners. Although in general the number of candidate schedules the partitioner must examine grows exponentially with graph size, the complexity is largely constrained 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 consumes more than 18 hours on a P3 500Mhz system, examining in excess of 101 million candidates.

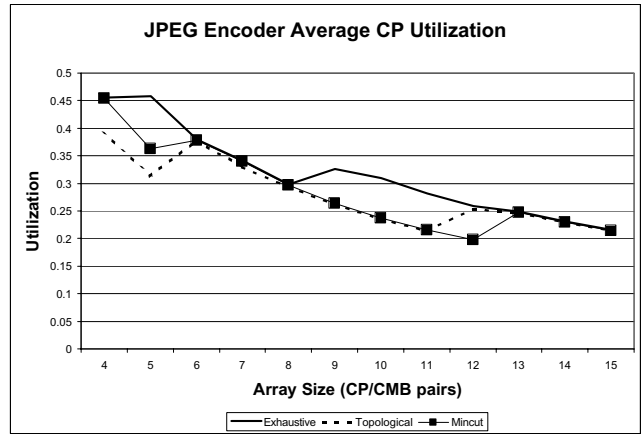


Figure 8: JPEG encoder: comparison of average CP utilization with three implemented partitioners.

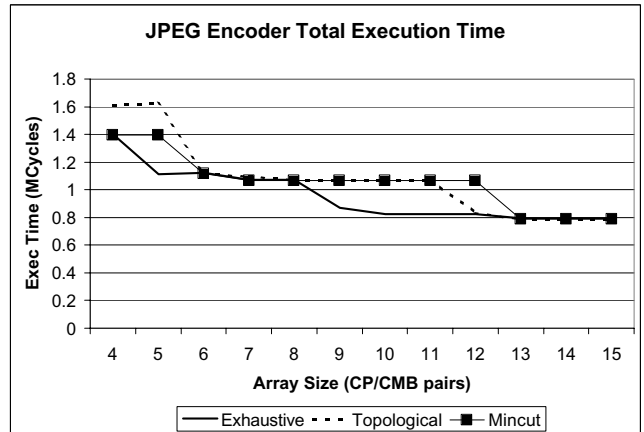


Figure 9: JPEG encoder: comparison of total execution time with three implemented partitioners.

Table 2 presents cost for each heuristic partitioning algorithm for varying graph size and target array size. Figure 8 summarizes partitioning results for the 13-page JPEG encoder application.

The plot in Figure 8 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 Balanced Mincut heuristic partitioners in *the worst case* perform within 17% of the optimal on JPEG encoder and other applications. Another unexpected result is that neither heuristic partitioner continuously outperforms the other, although they differ greatly in algorithmic complexity.

Figure 9 shows JPEG encoder total execution time using the different partitioners. The expected inverse relationship between the total execution time and the average CP utilization holds, validating our hypothesis that increased utilization is a good predictor of reduced execution time.

5.2 Buffer Allocation

The quasi-static scheduler uses *stall detect* (Section 4.4) to perform the *Timeslice Sizing* step and to determine the amount of time each subgraph executes. *Buffer sizing* is the primary mechanism by which the static schedule generator controls the *Timeslice Sizing*, because the buffer size directly determines the “amount of work” a given subgraph can per-

Cost of Partitioning Algorithms (millions of CPU cycles)

App Name	V	E	min size	Balanced Mincut			Topological		
				min	25%	50%	min	25%	50%
JPEG enc	20	54	4	3.0	3.4	0.9	0.6	0.7	0.7
JPEG dec	16	56	3	1.8	1.1	1.3	0.3	0.3	0.3
Wavelet enc	36	56	6	3.3	2.1	4.8	0.5	0.5	0.5
Wavelet dec	33	51	6	4.4	3.4	5.5	0.3	0.3	0.3

Table 2: Time cost for computing a partition set using Balanced N-way Mincut and Topological partitioners. For each partitioner, we show the application-specific costs for three array sizes: minimum feasible (min), quarter spatial (25%), half spatial (50%).

form before stalling and forcing a reconfiguration. The current implementation of the quasi-static scheduler allocates the same amount of memory for all buffers, frequently resulting in under-utilized CMB memory and leading to needless data transfers between the array and the primary memory. We are currently working on a solution that allocates buffer sizes proportional to application requirements and attempts to balance CMB memory utilization with reconfiguration time and limited off-chip bandwidth.

6. ANALYSIS OF RESULTS

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

Figure 10 shows the total execution time of the JPEG decoder application (12 pages) for various array sizes, using the dynamic and quasi-static schedulers. We show two sets of curves: (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 2 on average. Similar results were observed for the JPEG encoder, the wavelet encoder, and wavelet decoder, with speedups of 2.2 to 5.7 (see Table 3). In addition, we find that the scheduling overhead (*i.e.* the difference between the overhead and no-overhead curves) is dramatically smaller for the quasi-static scheduler than for the dynamic one, typically by a factor of 10.

Figure 11 shows the total execution time of the JPEG decoder, with scheduling overhead, for several schedulers. The so-called *static* scheduler is a simple variation on the quasi-static scheduler where we disable stall detection, so as to use only fixed time-slices. This variation represents point 2 in Figure 4, a more static 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 the performance of the quasi-static scheduler. Interestingly, the 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

the feasible scheduling granularity.

In summary, improvement in the scheduling quality of the quasi-static scheduler can be attributed to several factors:

- the *global* view of the graph topology, instead of limited “frontier” (BFS) in the dynamic scheduler,
- using application execution profile to *predict* graph behavior, making an educated guess, instead of dynamically adapting scheduling decisions based on stale array state, and
- *fine-grained* hardware supported *stall detect* to automatically adapt *Timeslice Sizing* to dynamic changes in an application, rather than using a fixed time-slice as in the dynamic scheduler.

7. FUTURE WORK

To date, this work has not addressed the issue of low hardware utilization using the quasi-static scheduler. A close analysis of curves on Figure 8 shows that in the ideal (no overhead) simulation the average CP utilization varies from 47% in small to 20% in large array. Hence, over half of the potential computing capacity of the array is lost to CP stall and idle cycles. This is largely an artifact of co-scheduling neighbor nodes with *mismatched* I/O rates. We are working to address it (1) in the scheduler with techniques to insert buffers to decouple rate mismatched nodes, and (2) in the compiler with transformations (*e.g.* arithmetic parallelization/serialization) that attempt to match rates between neighbor nodes.

Neither the dynamic nor the quasi-static schedulers considers the effects of routing and placement. Thus far, the implementations assumed an “ideal” array; we plan to add a page placement and routing engine in the near future. While routing may not be a significant problem in small arrays where *crossbar* switches are feasible, larger arrays demand a more economical switching structure and require taking advantage of locality in the computation graph.

8. CONCLUSION

SCORE attempts to eliminate barriers to efficient exploitation of reconfigurable devices by using a paged virtual hardware model that enables application compatibility, longevity and automatic performance scaling on larger hardware. The run-time scheduler plays a key role by supporting the abstractions of SCORE that gives a programmer expressive power and flexibility. A poorly implemented scheduler may drastically diminish the potential performance gains and severely limit the model’s applicability.

We presented a taxonomy of scheduling solutions to demonstrate that, in addition to the natural solution of a fully

	App Size (pages)	Realistic Simulation		Ideal Simulation	
		Quasi-Static	Static	Quasi-Static	Static
JPEG enc.	13	2.8	1.3	2.2	1.0
JPEG dec.	12	2.5	1.3	2.0	1.0
Wavelet enc.	30	4.5	1.6	3.3	1.1
Wavelet dec.	27	7.3	3.0	5.7	2.3

Table 3: Average reductions in application total execution time relative to the Fully Dynamic Scheduler.

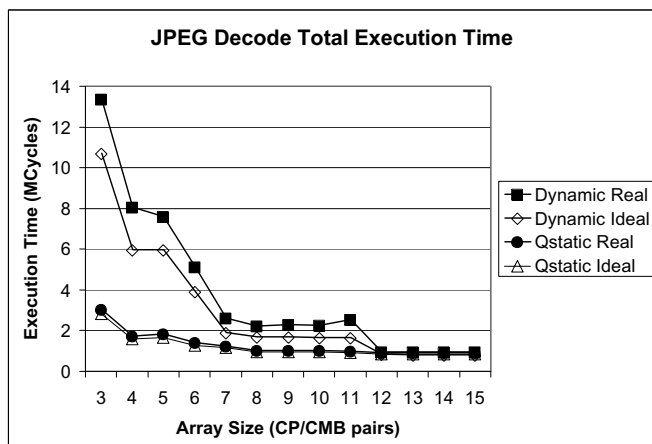


Figure 10: JPEG decoder: Total execution time comparison between quasi-static and dynamic schedulers for both ideal and realistic array simulation.

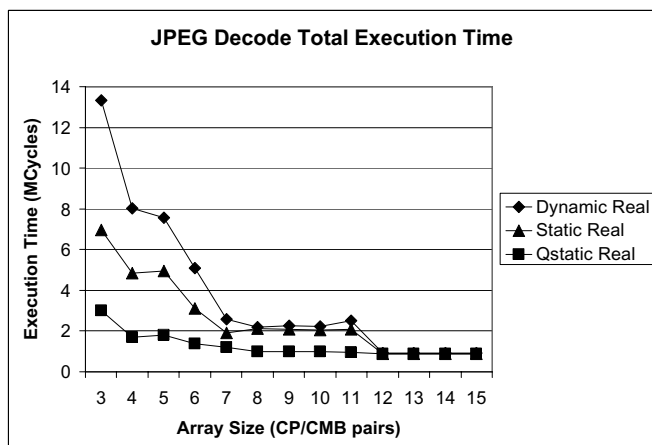


Figure 11: JPEG decoder: Total execution time comparison between the fully dynamic, fully static and the quasi-static scheduler obtained from a realistic array simulation.

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.

We demonstrated two scheduling approaches and found the quasi-static approach to have superior schedule quality and substantially lower run-time overhead than the dynamic approach. Our quasi-static implementation reduced run-time scheduling overhead by an average factor of 10, down to 10,000 cycles per time-slice. The quasi-static implemen-

tation also reduced execution times by an average factor of 3 for a set of applications containing both static and data-dependent components. Without any code change or recompilation, all applications execute using either scheduler.

9. REFERENCES

- [1] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.
- [2] J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token flow Model*. PhD thesis, UC Berkeley, 1993.
- [3] J. 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.
- [4] E. Caspi, M. Chu, R. Huang, N. Weaver, J. Yeh, J. Wawrzynek, and A. 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.
- [5] M. M. Chu. Dynamic runtime scheduler support for score. Master's thesis, UC Berkeley, 2000.
- [6] A. DeHon. *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis, MIT, 545 Technology Sq., Cambridge, MA 02139, September 1996.
- [7] S. Ha. *Compile-Time Scheduling of Dataflow Program graphs with Dynamic Constructs*. PhD thesis, UC Berkeley, 1992.
- [8] 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.
- [9] S. Ha and E. A. Lee. Compile-time scheduling of dynamic constructs in dataflow program graphs. *IEEE Transactions on Computers*, 46(7), July 1997.
- [10] E. Lee. *Advanced Topics in Data-Flow Computing*, chapter Static Scheduling of Data-Flow Programs for DSP, pages 501 – 527. Prentice-Hall, Inc., 1991.
- [11] H. 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.
- [12] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, UC Berkeley, 1995.