

# Scheduling on a Reconfigurable Processor with Virtual Pages

CS-262 Course Project  
Prof. Eric Brewer  
University of California, Berkeley

Eylon Caspi, Michael Chu, Randy Huang

{eylon,mmchu,rhuang}@cs.berkeley.edu  
<http://www.cs.berkeley.edu/~eylon/cs262/>

December 15, 1998

## Abstract

Reconfigurable computing devices offer substantial improvements in functional density and yield versus traditional microprocessors, yet remain out of general-purpose use due in part to their difficulty of programming and lack of cross-device compatibility. This project presents a reconfigurable architecture which supports automated compilation and cross-device compatibility by virtualizing hardware resources and automating management thereof in operating system services. A study of architectural parameters indicates that this architecture would benefit from small time slices on the order of a millisecond, and from a high availability of configurable memory resources. A dynamic scheduling algorithm is presented whose effectiveness is found to depend heavily on clustering and co-scheduling of virtual compute operators.

## 1 Introduction

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

Reconfigurable devices have proven extremely efficient for certain types of processing tasks. The

key to their cost/performance advantage is that conventional processors are often limited by instruction bandwidth and execution restrictions or by an insufficient number or type of functional units. Reconfigurable logic can exploit more program parallelism. By dedicating significantly less instruction memory per active computing element, reconfigurable devices can achieve a 10x improvement in functional density over microprocessors. At the same time this, lower memory ratio allows reconfigurable devices to deploy active capacity at a finer grained level, allowing them to realize a higher yield of their raw capacity, sometimes as much as 10x versus conventional processors [DeHon96], [DeHon98].

Despite the advantages in cost/performance, reconfigurable devices exist mainly as application-specific devices, out of reach of typical software programmers. One of the reasons is the lack of convenient programming tools and environment. The current design methodology for these devices typically requires programming in high-level hardware description language like Verilog or VHDL instead of a more familiar programming language like C or Java. Another reason preventing reconfigurable devices from gaining acceptance is that attaining their performance/cost advantages often requires exposing the underlying hardware to the programmer, making user programs device-dependent.

The main research goal of the Berkeley Reconfigurable Architecture, System and Software group (BRASS) is to explore reconfigurable devices as general-purpose computation devices. A general-purpose device must:

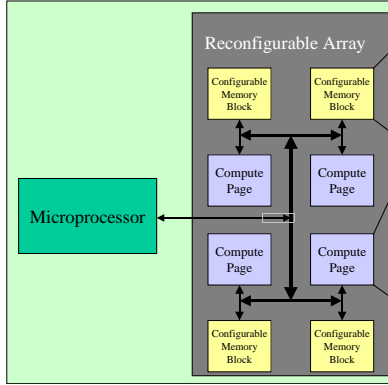


Figure 1: BRASS reconfigurable microprocessor, block diagram

- solve the entire problem, not just computational kernel
- graciously handle heavy memory demands
- support a convenient programming environment
- allow software to survive from one generation to the next

To realize these goals, the BRASS group must address the following questions:

- what is the programming model used to develop software?
- what architecture features are required to support this programming model?
- what is the interface between array and memory?
- what is the interface between array and microprocessor?
- what is the application binary interface (ABI) which would allow software to survive?

This paper represents a continued effort by the BRASS group to answer some of these questions. In particular, the project described herein aims to reduce programmer effort in managing large computations on the BRASS reconfigurable processor (Figure 1) by providing an abstraction of virtual compute pages analogous to traditional virtual memory pages. In such a scheme, a programmer (or CAD tool) would partition a computation into fixed-size virtual computational units but leave their scheduling onto physical units to a run-time operating system. Hiding the total size of physical resources would

relieve the programmer from resource size constraints as well as provide binary compatibility across a family of different-sized reconfigurable arrays.

The BRASS reconfigurable processor integrates a microprocessor, DRAM, and a reconfigurable array on a single silicon device. The schedulable unit of processing on the array is a Compute Page (CP), consisting of fine-grained, programmable computational elements. Memory on the array is partitioned into Configurable Memory Blocks (CMB) which provide DRAM storage to CPs using random-access or streaming mechanisms. These elements of the array communicate over a programmable, hierarchically-connected network.

This project is divided into six parts:

1. Define preliminary details of a programming model
2. Select and map several applications in this programming model
3. Build an array simulator
4. Build a centralized scheduler
5. Try different scheduling algorithms and heuristics
6. Understand architectural tradeoffs

The paper is organized as follows. The programming model and selected applications mapped to the reconfigurable array are described in section 2. Issues in implementing the array simulator and the scheduler are discussed in sections 3 and 4. Analysis, results and observations are presented in section 5. Related work is discussed in section 6. The paper concludes with future work and a conclusion in sections 7, and 8.

## 2 Programming Model

The programming model chosen for expressing computation in this project is a restricted form of the proposed SCORE (*Stream Computations Organized for Reconfigurable Execution*) programming model [DeHon98]. SCORE is a stream-based model influenced strongly by dataflow architectures (see related work in section 6). A computation is specified by a dataflow graph containing computational operators

which communicate via streams of data tokens. The graph makes producer-consumer relations explicit. Individual operators can be made to run in parallel, forming token-based pipelines. Using automatic compilation techniques, a computation expressed in this model could be made to run on any of a variety of hardware resources, with operating-system or hardware supported scheduling of the computational operators.

In the incarnation of SCORE used for this project, operators are clustered into CPs for the BRASS reconfigurable processor. Streams are implemented directly in the pipelined network hardware. We restrict the consumption rates of input tokens in any page to have fixed ratios. That is, a page acts as a macro-operator which fires whenever some known-size set of input tokens becomes available. We allow the rate of output token production to be data dependent. Each stream may have only one producer and one consumer, so that signal fanout is avoided. For instance, a CMB may connect to exactly one CP at a time. Furthermore, the dataflow graph of a computation may not be dynamically altered.

In order for the authors to understand the programming interface requirements, 3 application kernels were coded into behavioral descriptions using the restricted SCORE model. The kernels, extracted from the UCLA MediaBench suite [LPMS97] and the Honeywell Stressmark suite [Hone97] included ADPCM, Gouraud-shaded triangle rasterization, and wavelet transform. ADPCM (*Adaptive Differential Pulse Code Modulation*) is a signal compression algorithm typically used on speech. Gouraud-shaded triangle rasterization is a key operation in 3D graphics rendering. Wavelet transform is a fundamental component of certain image compression algorithms. The experience of programming these kernels helped formalize the SCORE semantics for data-dependent token rates and end-of-stream conditions. While data-dependent output rates were easy to support, data-dependent input rates made it difficult to determine when a page operator is ready to fire and should be swapped into the array.

To better illustrate the SCORE programming model, Figure 2 shows an example of how a page operator and its inter-page streams are specified and connected. The example depicts the composition of two two-input adders to form a three-input adder. The entire specification is written in Java by subclassing certain primitive objects.

A page operator is specified by subclassing the

```

1 public class Adder extends ComputePage {
2   PageInput  a=new PageInput(this,8,1);
3   PageInput  b=new PageInput(this,8,1);
4   PageOutput o=new PageOutput(this,8);
5
6   public Adder() {
7     betweenFirings(1);
8   }
9
10  public int step() {
11    long aValue=a.dequeue();
12    long bValue=b.dequeue();
13
14    if ((aValue==0)&&(bValue==0)) {
15      signalDone(4);
16    }
17    o.queue((aValue+bValue),4);
18    return(1);
19  }
20
21 public class Add3 extends Design {
22   CMBSeqSrc aSrc=new CMBSeqSrc();
23   CMBSeqSrc bSrc=new CMBSeqSrc();
24   CMBSeqSrc cSrc=new CMBSeqSrc();
25   Adder add1=new Adder();
26   Adder add2=new Adder();
27   CMBSeqSink oSink=new CMBSeqSink();
28
29   public Add3() {
30     connect(aSrc.dataOut,add1.a);
31     connect(bSrc.dataOut,add1.b);
32     connect(add1.o,add2.a);
33     connect(cSrc.dataOut,add2.b);
34     connect(add2.o,oSink.dataIn);
35   }
36 }

```

Figure 2: Sample program for behavioral SCORE: composing 2 2-input adders into a 3-input adder

`ComputePage` class (*line 1*). The first thing to specify in an operator are its inputs and outputs (*lines 2-4*). Each input and output has a specified a bit width (e.g 8 bits). Inputs also have a specified consumption rate per operator firing (e.g. 1 token per firing). The length of an operator firing may be specified statically (as in *line 7*) or remain fully dynamic. For fully pipelined operators, firing time should be 1 cycle. The firing behavior of an operator is specified by its `step()` method, which is responsible for consuming input tokens and producing output tokens. Output tokens are queued with a specified cycle-delay (*line 17*) meant to model the computational delay

of hardware. Operators detect end-of-stream conditions in using any user-defined mechanism, e.g. receiving an invalid token value, or receiving 0 on a dedicated control stream. Upon detecting end-of-stream, an operator raises a “done” signal to the array controller using the `signalDone` method (*line 15*), specifying an output delay sufficient to flush its internal pipeline. The firing method `step()` normally returns the number of cycles taken in each particular firing. This time is typically 1 for pipelined operators, and may be dynamic, but must be no smaller than any of the specified output delays.

A computation combining multiple operators is specified by sub-classing the `Design` class (*line 21*). After the CP and CMB page operators are instantiated (*lines 22-27*), inter-page streams may be connected using the `connect()` method (*lines 30-34*).

## 3 Architectural Simulation

### 3.1 Behavioral Simulator

A behavioral, event-driven simulator was developed to model timing of computation and communication on the reconfigurable array. Every action on the array, including page firings, configuration loads, token communication, etc., is described by a start and completion event pair. A master event queue is used to process events in correct order. Event handlers, used to simulate each action in turn, typically install subsequent events to happen at future times. For example, the start event for a page firing executes the page’s `step()` behavioral code, which may install token emission events. The completion event of the page firing either installs the next firing event or manages stalling the page. The behavioral code for a page firing need not be pre-empted should the firing stall on a full output buffer. The events installed by that code will simply be delayed until the page unstalls.

### 3.2 Hardware Model

The array simulator implements a particular hardware architecture with several parameters. The array is assumed to contain an arbitrary number of frames, each containing a single CP and CMB. While CPs are user-programmable, CMBs are assumed to operate in one of three OS-controlled modes:

- read/write RAM
- reading sequential memory
- writing sequential memory

Memory management is supported by equipping each CMB with a pair of segment bound registers and an asynchronous address-fault signal for segment violations.

Pages communicate using pipelined token streams on point-to-point paths allocated in a programmable network of arbitrary geometry. At present, we assume a binary fat-tree network with flight-time proportional to the number of tree levels crossed. We ignore any routing conflicts. The network is assumed to support data-presence and back-pressure signals, allowing the communication rate on a stream to vary dynamically. Hardware pages are self-timed by the availability of input tokens and buffer space for output tokens. When firing, a page may send asynchronous signals to the array controller for attention-request. Signals presently supported include an address-fault signal for CMBs, a “done” signal for CPs, and a stall “give-up” signal raised by pages which have been stalled for longer than some threshold time period.

Communication between the array and the micro-processor exists in several flavors. Processor commands to the array include page start/stop directives, context swaps, CMB-to-CMB block memory transfers, and DMA-based streaming memory transfers in/out of the array. Commands from the processor are issued asynchronously and may complete out of order. The array immediately reports to the processor all page signals and completions of context swaps and memory moves (physically, this could be implemented by interrupts or processor polling). The present scheduler implementation queues each report and services the array only on timer interrupts.

## 4 Scheduling

The scheduler written for this project is composed of 4 main steps:

- `buildRQ()`: Build a ready-queue of clustered schedulable operators,
- `buildPriorityRQ()`: Prioritize the ready-queue,

`jobSelection()`: Map operators to available physical CPs and CMBs,

`issueCommands()`: Issue a sequence of commands to the array (simulator) to configure pages and to connect/disconnect streams.

The scheduler begins by building a ready-queue of schedulable operators. An operator is considered to be schedulable if:

1. for each of its inputs:
  - (a) the predecessor operator is resident on the array or in the ready-queue, or
  - (b) sufficient input tokens are queued up,
2. for each of its outputs:
  - (a) there is sufficient output buffer space.

The method `buildRQ()` traverses a waiting-queue of operators waiting to be scheduled and determines based on the above criteria whether each one is schedulable. Schedulable operators are moved to the ready-queue as soon as they are identified. For stronger clustering, `buildRQ()` can optionally check if each successor of a newly-identified schedulable operator is itself schedulable, and to move successors to the ready-queue before continuing to process the waiting-queue. This clustering scheme causes neighboring operators from the dataflow graph to be inserted together into the ready-queue, increasing their chance to be co-scheduled on the array. A weaker clustering can still be achieved without chasing successors, by merely checking each waiting operator in turn. This scheme, though it is less computationally expensive, may miss and delay scheduling of otherwise schedulable operators, or even cause deadlock.

After building a clustered ready-queue, `buildPriorityRQ()` can choose to prioritize operators in the ready-queue. The intent of prioritizing is to more effectively choose which operators should be run next in the array under constrained physical resources. Operator priorities might be based on the number of input tokens queued up, the number of successors in the dataflow graph, whether the operator is already loaded on the array, or on any other cost. In the current implementation, no prioritizing is done. The reason for this is that prioritizing tends to break the dynamic clusters performed in `buildRQ()`, and in experiments, often leads to poor runtime performance.

After optional prioritizing, the ready-queue is passed to the `jobSelection()` method, whose job is to assign ready operators to physical pages on the array. The assignment should strive to perform as much computation as possible in the coming time slice. Many methods and metrics exist for optimizing the assignment. The simplest way to assign operators is to assign them from the top of the ready queue to the next free CP on the array, until no free CPs remain. If the successor of an assigned operator cannot be co-scheduled with that operator, then a “stitch” CMB<sup>1</sup> is allocated to capture the output stream, and to save its tokens until the successor is run in the array and can accept them.

One problem with this technique is what to do when the number of CMBs needed for stitching exceeds the number of available CMBs on the array? One possibility is to simply remove from the schedule any operators whose outputs cannot be stitched. However, if a removed operator has many input streams from other scheduled operators, then stitching those streams could lead to an even greater CMB requirement. The naive approach to this problem would require checking all successors and predecessors of a page and backtracking out of impossible schedules. Formally, this problem could be stated as a min-cut problem on the schedulable subset of the dataflow graph. Neither case seems to have a particularly easy solution.

Instead, our implementation of the scheduler performs simple, greedy trials. In each trial, it attempts to add one operator from the ready-queue to the array. It calculates the number of remaining free CMBs by adding stitch CMBs for any of the operator’s successors which are not scheduled and subsequently adding back any stitch CMBs not needed by scheduled predecessors of the operator. As each compute operator is added in a new trial, the tally of remaining CMBs may increase or decrease, and is allowed to temporarily become negative. The order of operator additions is recorded. When no free CPs remain on the array, `jobSelection()` chooses which operators to discard by simply reverting to the last trial which did not require more CMBs than are available in the array.

After assignment of operators to physical pages, the final phase `issueCommands()` issues commands to the array (simulator) so as to load the newly as-

---

<sup>1</sup>The name “stitch” CMB was chosen because such a CMB stitches together a data stream between two non-co-scheduled compute pages. We use “stitching” to mean capturing and retiming a disconnected stream

signed operators. Commands are grouped from a serial command stream into a sequence of maximally-parallel, hardware-feasible operations. For instance, the loading of a CP configuration from a CMB can be done simultaneously for many different CPs. The scheduler waits for completion notifications from the array before issuing each new group of commands. A scheduled page may begin operation as soon as its configuration is loaded and all its streams are connected.

The complexity of the scheduling algorithm is as follows. Let  $N$  be the number of physical CPs and  $M$  be the number of compute operators in the design. Then the complexity of the scheduling phases is:

- `buildRQ()`:  $O(M^3)$
- `buildPriorityRQ()`: (not performed)
- `jobSelection()`:  $O(NM^2 + NM)$
- `issueCommands()`:  $O(N)$

The total complexity is  $O(M^3 + NM^2 + NM + N)$ , or simply  $O(M^3)$  for designs significantly larger than the physical array.

## 5 Results and Analysis

A number of experiments were performed to assess the effectiveness of the scheduling algorithm and run-time system. Each experiment compares the effect of various architectural parameters on the makespan, or total run time, of three randomly-generated programs (a small, medium, and big design). Each program represents a pipelined computation on a single stream of data, described by a random dataflow graph with arbitrary depth and number of nodes. Table 1 lists the parameters of the random graphs. Parameters varied in the experiments include, for the array architecture:

- array size
- scheduling time slice
- CP to CMB ratio

and for the scheduling policy:

- enabling/disabling clustering

	Number of CPs	Graph depth
small	8	5
medium	15	10
big	25	18

Table 1: Specification of the random program graphs (Graph depth refers to the number of CPs on the longest path from input to output)

- effect of feedback loops in graphs

### 5.1 Array Size and Clustering

Figure 3 shows the effect on makespan as array size is varied from 5 to 32 frames (CP-CMB pairs), with weak clustering. In this configuration, the scheduler’s `buildRQ()` routine makes only a single pass over the waiting queue, without chasing dataflow successors. In general, makespan is seen to decrease monotonically with array size. The medium and big designs exhibit two distinct plateaus. The first plateau is due to insufficient physical CMBs for stitching multiple data streams, leading to underutilization of CPs and a high number of context swaps. The second plateau occurs when the array becomes large enough to fit the entire design, so no stitching or context swaps are needed. The poor performance of the small design is due to the naive single-pass `buildRQ()` algorithm failing to realize that all pages are ready to be scheduled, even if the array is large enough to hold them all. The few increases in makespan which violate monotonicity for the medium and big designs are similarly believed to be artifacts of the single-pass algorithm.

Figure 4 repeats the experiment of Figure 3 with strong clustering. In this configuration, the scheduler’s `buildRQ()` routine looks for successors of ready operators in the waiting queue, achieving the effect of a dynamic topological sort. The intended effect of stronger clustering is to reduce stitch-CMB usage by co-scheduling more pages which communicate with each other. The medium design benefits significantly, no longer exhibiting a CMB-limited intermediate plateau for makespan. The big design seems to benefit little from better clustering, seeing some improvement on small arrays. In addition, we find that the artifacts of the single-pass algorithm are no longer present. The small design sees a significant improvement from being correctly co-scheduled in sufficiently large arrays, and the medium and

big designs have smoother, monotonically decreasing makespans. Overall, makespan improvement from clustering seems to be highly graph-dependent and ranges from nil to significantly better.

## 5.2 Feedback in Dataflow Graphs

In dataflow graphs with feedback, data dependencies along a feedback cycle prevent pipelining of large blocks of data through any incomplete subset of the cycle. Thus to obtain an efficiently pipelined computation, all pages on a feedback cycle should be co-resident in hardware. Clearly, this is impossible on arrays with fewer frames than the maximum cycle size. Optimal pipelining may be impossible on larger arrays as well for graphs with multiple intersecting cycles. If all feedback cycles cannot be co-scheduled on the array, then multiple context swaps will be necessary as each token traverses the cycles. Feedback-free graphs (DAGs) are free of such restrictions and can efficiently pipeline any connected path in the graph. Hence, on small arrays, we should expect significantly better performance with feedback-free graphs than with feedback graphs of the same size.

Whereas Figures 3 and 4 represent the general case of graphs with feedback, Figure 5 repeats the experiment for strictly feed-forward graphs. The random graphs used are still sized as in Table 1 but are generated without feedback cycles. Weak clustering is employed, i.e. no sorting of the waiting queue. Comparing with the feedback graphs of Figure 3, we find that all three feed-forward graphs have significantly smaller makespans on small arrays, 3 to 8 times smaller on the minimum-sized 5-frame array. The makespan on arrays large enough to co-schedule each graph in its entirety is comparable to the feedback case. The big design clearly suffers from artifacts of the non-clustering, single-pass `buildRQ()` algorithm, whose choice of fireable pages to schedule is arbitrary. Due to lack of time, we could not repeat the feed-forward experiment with strong clustering.

## 5.3 Choice of Time Slice

Because scheduling decisions and context swaps occur only on timer interrupts, it is reasonable to suspect that the choice of time slice should affect overall performance. Figure 6 plots the effect of time-slice length on the makespan of the medium design on a

6-frame array. The minimum makespan at a time-slice of 2000 cycles coincides precisely with the dataset size of 2000 tokens. This is no coincidence, since a pipelined design processes 1 token per cycle, and each scheduled subset of the graph exhausts its inputs after 2000 cycles. With larger time slices, pages on the array waste cycles waiting for service after exhausting their inputs. With smaller time slices, frequent scheduling decision add dominating overhead. Note that the present implementation of the scheduler does not pre-empt pages which are not stalled. Naive pre-emption of active pages would add even more overhead for time-slices smaller than the dataset size.

While data-sets may be arbitrarily large, the useful data size in any scheduling run is physically limited the size of each CMB. Hence, for large data-sets, the choice of CMB size would directly affect the optimal time-slice length. The presently proposed 2Mbit CMB could contain, for example,  $2^{18}$  (256K) 8-bit tokens,  $2^{17}$  (128K) 16-bit tokens, or  $2^{16}$  (64K) 32-bit tokens. At the proposed 250Mhz clock cycle, assuming 1 token processed per cycle, a CMB would be exhausted in 0.25ms to 1ms, and context swaps would be needed 1000 to 4000 times per second. Such time-slices are significantly smaller than in conventional microprocessor-based multitasking operating systems, which typically time-slice 60 to 100 times per second. To support such high interrupt rates, a host processor may be required to run no programs other than the array scheduler so long as the array is active. This possibility raises some interesting fairness issues for multitasking operating systems. It may be appropriate to require that jobs be submitted to the array synchronously, so that a user program's time slice is spent either on the processor or on the array, but not on both. In addition, it will be important to reduce reconfiguration times to be on the order of a time-slice or less, lest they begin to dominate actual computation time.

## 5.4 CMB to CP Ratio

Figures 3 and 4 indicate that designs run on small arrays may suffer from a shortage of CMBs for stitching (see 5.1). The reason for this is that a CMB has only one stream controller and can thus support only one stitched stream at a time. It may be desirable, therefore, to equip an array with more CMBs than CPs. Table 2 shows the effects of increasing the CMB-to-CP ratio in an 8-CP array for each of the

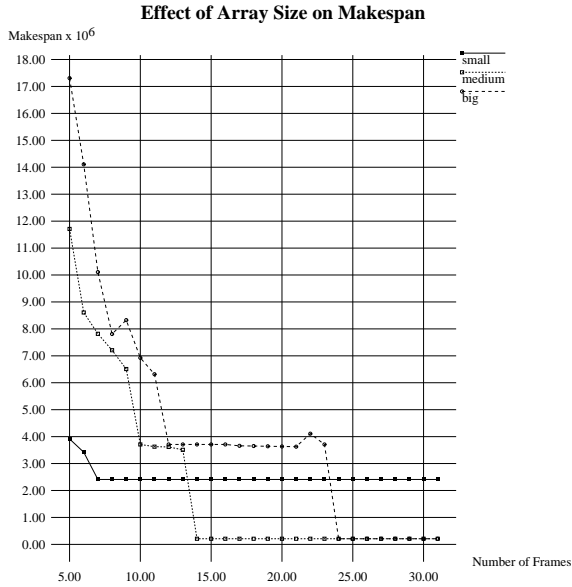


Figure 3: Effect of array size on makespan, weak clustering

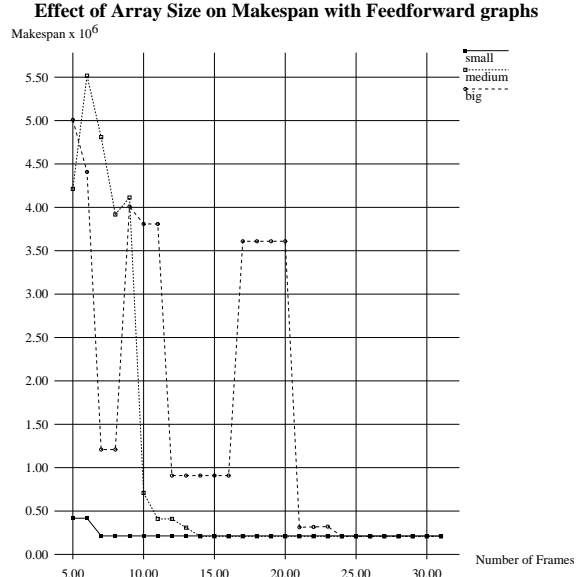


Figure 5: Effect of array size on makespan for feed-forward graphs

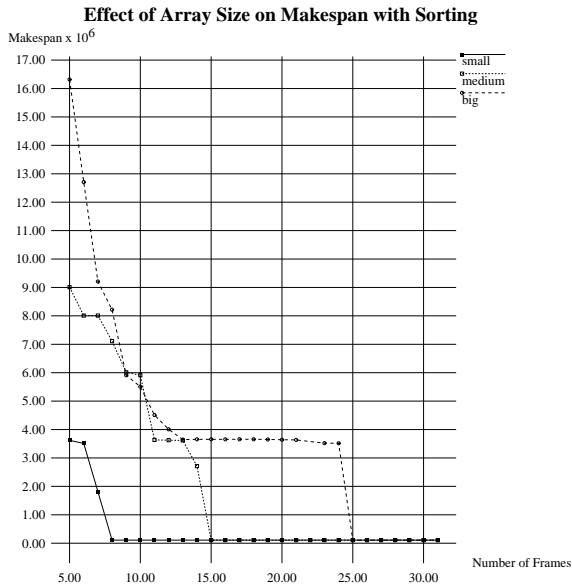


Figure 4: Effect of array size on makespan, strong clustering

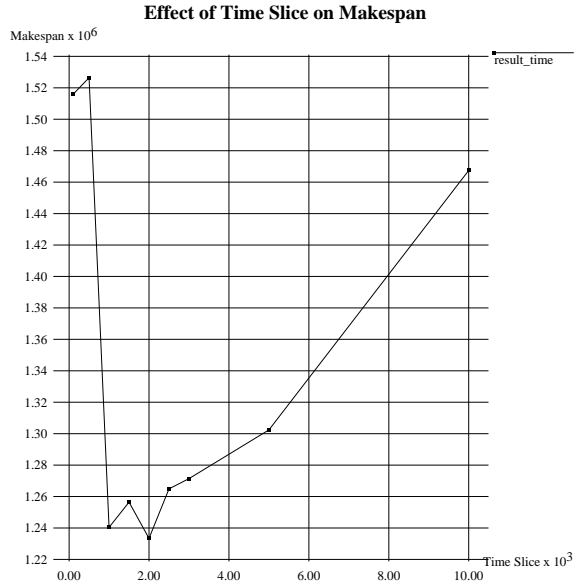


Figure 6: Effect of Time-Slice on Makespan for a 6-frame array

three random graphs<sup>2</sup>. While the big and small design obtain little or no benefit from additional CMBs, the medium design gains a twofold increase in speed from doubling the number of CMBs. Increasing the number of CMBs by more than double does not help any of the designs. Note that an array size of 8 CPs is below the CMB-limited makespan plateau of figures 3 and 4. It would be interesting to create analogous graphs of makespan versus array size for the case of doubled CMB count, and to observe the effect on those plateaus.

	$\frac{CMB}{CP} = 1$	$\frac{CMB}{CP} = 2$	$\frac{CMB}{CP} = 4$
small	2.4	2.4	2.4
medium	7.2	3.7	3.7
big	7.8	6.7	6.7

Table 2: Makespan (in *millions* of cycles) for various CMB-to-CP ratios on an 8-CP array

## 6 Related Work

### 6.1 Multiprocessors

The BRASS reconfigurable processor shares with the multiprocessor community the notions of *priority-list scheduling* [GDWL92] and *gang scheduling* [FPRu96]. Given a task precedence graph, priority-list scheduling uses a priority function to choose, from among all tasks whose predecessors have completed, which to schedule. In the SCORE model, priorities can be used in conjunction with a dataflow graph to choose among operators whose predecessors have fired. Priorities may include, for instance, the number input tokens queued, and whether an operator configuration is already loaded on the array. [LAAG94] found that no single priority heuristic was optimal across different program structures and multiprocessor configurations, but that adaptive combinations thereof produced good results. Gang scheduling involves co-scheduling related tasks. In the SCORE model, it is clearly advantageous to co-schedule neighboring operators from the dataflow graph. In addition, it is highly advantageous to co-schedule all operators belonging to a feedback loop, in order to avoid context swaps in each traversal of the loop.

<sup>2</sup>Although the array simulator assumes a fixed CMB-to-CP ratio of 1, it is possible to vary the effective ratio by increasing array size while restricting the number of CMBs usable by the scheduler

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

### 6.2 Dataflow Systems

Because fully-dynamic, run-time scheduling can be prohibitively expensive, various efforts appear in multiprocessing literature to exploit compile-time scheduling [KKTa90], [YTYa95]. Such efforts typically assume a fixed or highly-predictable communication structure among known computational elements. Such restrictions are well modeled by *dataflow* computational models, in which a computation is described by the flow of tokens along a graph of computational operators, without explicit control structure. The SCORE model is essentially a dataflow on CP-sized macro-operators, each of which clusters traditional dataflow operators (specifically, integer-controlled dataflow operators in our restricted SCORE model).

Synchronous Dataflow (SDF) is a dataflow computational model in which the number of tokens consumed and produced in each firing of an operator is known at compile time. SDF is thus amenable to static scheduling with minimal run-time overhead. Although SDF is not Turing-complete due to lack of conditional control, it is sufficient for many digital signal processing tasks (e.g. FIR/IIR filtering). A theoretical framework exists for statically scheduling SDF graphs on uniprocessors [BMLe96], which can find (or disprove the existence of) periodic firing schedules with guaranteed

memory requirements and deadlock-free operation. Boolean-controlled Dataflow (BDF or Token Flow, [BuLe92], [BuLe93], [Buck93]) and Integer-controlled Dataflow (IDF, [Buck94]) are Turing-complete extensions of SDF which add simple conditional operators. Scheduling of BDF and IDF graphs on uniprocessors typically requires clustering subgraphs to run in successive phases, so as to bound memory requirements.

Scheduling dataflow graphs on parallel hardware has additional synchronization complications due to (i) heterogeneous operator firing times, (ii) network delays, and (iii) clustering of operators on processors. [WiLe96] implements a mapping of SDF to VHDL for hardware generation, where the creation of arbitrary control and synchronization signals obviates the need for operator clustering. With regards to clustering on conventional multiprocessors, there has been some work in compile-time scheduling based on runtime profiles [HaLe97] as well as static graph analysis [BSLe95], [PBLE95]. Fully-dynamic scheduling, due to its high cost, is typically not the best solution in computational domains which have static guarantees, such as SDF. [Lee91] defines a taxonomy and discusses tradeoffs in the spectrum between fully-static and fully-dynamic dataflow scheduling.

[JoVa96] describes a heuristic, on-line, SDF scheduling algorithm for idealized message-passing multiprocessors similar in some respects to the BRASS reconfigurable processor. The algorithm exploits pipelining by scheduling “linear clusters” of dataflow operators. Each node in such a cluster has exactly one dataflow successor in the cluster, so the nodes form a pipeline for tokens. The study reports 80-90% utilization in the used processors for several feed-forward applications and 10% utilization for a feedback-constrained application. The study does not discuss memory constraints for data streams entering or leaving clusters, so it is possible that the utilization reported is high due to large or infinite memory assumptions.

## 7 Future Work

Planned extensions to the array simulator include presently-ignored costs such as finite propagation time of back-pressure signals, as well as presently-ignored restrictions such as finite network wires. A full treatment of network resources would require the addition of sophisticated online routing algorithms to the scheduler. The scheduler does not presently

guarantee fairness among multiple program graphs and would benefit from such techniques as lottery-scheduling for fair multitasking. The fully-dynamic scheduling algorithm described in section 4 is expensive and must be optimized to support the short time-slices suggested in section 5.3. Specifically, we would like to explore offline clustering algorithms to identify paths and cycles of dataflow graphs which should be co-scheduled. Virtual memory management to support co-location of multiple, smaller data streams in each CMB is being considered, in part for the purpose of reducing memory moves and context swaps needed for feedback-dominated computations.

## 8 Conclusion

We have presented a study of architectural parameters and scheduling policy for a reconfigurable device with virtual compute resources supported by the SCORE dataflow programming model. Work presented indicates that such a reconfigurable device needs fairly short time-slices and availability of more streaming memory resources than compute resources. We have found path-based clustering of communicating operators to be important in reducing program makespan, but potentially difficult and expensive to implement well. The interaction of clustering with priority-list scheduling is non-trivial, and our work suggests that prioritizing after clustering may actually degrade performance.

## References

- [DeHon96] André DeHon, *Reconfigurable Architectures for General-Purpose Computing*, AI Technical Report 1586 (Ph.D thesis), MIT Artificial Intelligence Laboratory, 545 Technology Sq., Cambridge, MA 02139, October 1996. <http://www.ai.mit.edu/people/andre/phd.html>
- [HSRA97] André DeHon, *HSRA Overview*, 1997. [http://www.cs.berkeley.edu/~amd/CS294F98/handouts/hsra\\_aug98.ps](http://www.cs.berkeley.edu/~amd/CS294F98/handouts/hsra_aug98.ps)
- [DeHon97] André DeHon, “Multicontext field-programmable gate arrays,” draft, 1997. [http://www.cs.berkeley.edu/~amd/CS294F98/papers/dpga\\_cs294.ps](http://www.cs.berkeley.edu/~amd/CS294F98/papers/dpga_cs294.ps)

- [DeHon98] André DeHon, "SCORE stream computations organized for reconfigurable execution," U.C.Berkeley BRASS internal draft, September 1998.
- [LRSa93] C.E.Leiserson, F.M.Rose, J.B.Saxe, "Optimizing synchronous circuitry by retiming," *Third Caltech Conference On VLSI*, pp. 87–116, March 1993.
- [LPMS97] C.Lee, M.Potkonjak, W.Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications Systems", *Proc. 30th Annual Intl. Sym. on Microarchitecture (MICRO-30)*, December 1997
- [Hone97] Honeywell, *Adaptive Computing Systems Benchmarking (Stressmarks)*, 1997-98. <http://www.htc.honeywell.com/projects/acsbench/>
- [BuLe92] Joseph T. Buck, Edward A. Lee, "The token flow model," presented at *Data Flow Workshop*, Hamilton Island, Australia, May 1992.
- [BuLe93] Joseph T. Buck, Edward A. Lee, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," *Proc. ICASSP'93*, Minneapolis, Minnesota, April 1993.
- [Buck93] Joseph T. Buck, *Scheduling Dynamic Dataflow Graphs With Bounded Memory Using the Token Flow Model*, Technical Memorandum UCB/ERL M93/69 (Ph.D thesis), EECS Dept., University of California, Berkeley, 1993.
- [Buck94] Joseph T. Buck, "Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams," *Asilomar Conf. on Signals, Systems, and Computers*, Pacific Grove, CA, Oct. 30–Nov. 2, 1994.
- [HaLe97] Soonhoy Ha, Edward A. Lee, "Compile-Time Scheduling of Dynamic Constructs in Dataflow Program Graphs," *IEEE Trans. on Computers*, vol. 46, no. 7, July 1997.
- [MiLe97] Takashi Miyazaki, Edward A. Lee, "Code generation by using integer-controlled dataflow graph," *Proc. ICASSP'97*, Munich, Germani, April 1997.
- [Lee91] Edward A. Lee, "Static scheduling of dataflow programs for DSP," Chapter 19 in *Advanced Topics in Data-Flow Computing*, ed. J-L.Gaudiot and L.Bic, Prentice Hall, Englewood Cliffs, New Jersey, 1991.
- [GDWL92] D.D.Gajski, N.D.Dutt, A.C-H.Wu, S.Y-L.Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, Boston, Massachusetts, 1992.
- [BMLe96] S.S.Bhattacharyya, P.K.Murthy, E.A.Lee, *Software Synthesis From Dataflow Graphs*, Kluwer Academic Publishers, Norwell, Massachusetts, 1996.
- [WiLe96] Michael C. Williamson, Edward A. Lee, "Synthesis of parallel hardware implementations from synchronous dataflow graph specifications," *Asilomar Conf. on Signals, Systems, and Computers*, Pacific Grove, CA, 1996.
- [Lee97] Edward A. Lee "A denotational semantics for dataflow with firing," Technical Memorandum UCB/ERL M97/3, Electronics Research Laboratory, Berkeley, CA 94720, January 15, 1997.
- [PBL95] J.L.Pino, S.S.Bhattacharyya, E.A.Lee, "A hierarchical multiprocessor scheduling system for DSP applications," *Asilomar Conf. on Signals, Systems, and Computers*, October 1995.
- [BSLe95] S.S.Bhattacharyya, S.Sriram, E.A.Lee, "Minimizing synchronization overhead in statically scheduled multiprocessor systems," *Proc. Intl. Conf. on Application Specific Array Processors*, 1995.
- [BSLe97] S.S.Bhattacharyya, S.Sriram, E.A.Lee, "Optimizing Synchronization in Multiprocessor DSP Systems," *IEEE Trans. on Signal Processing*, vol. 45, no. 6, June 1997.
- [JoVa96] Jan Jonsson, Jonas Vasell, "Real-time scheduling for pipelined execution of data flow graphs on a realistic multiprocessor architecture," *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, Atlanta, Georgia, May 7–10, 1996.
- [KKTa90] K.Konstantinides, R.T.Kaneshiro, J.R.Tani, "Task allocation and scheduling models for multiprocessor digital signal processing," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. 38, no. 12, December 1990.
- [LAAG94] G.Liao, E.R.Altman, V.K.Agarwal, G.R.Gao, *Proc. 27th Annual Hawaii Intl. Conf. on System Sciences*, 1994.
- [YTYa95] C.Yen, S.S.Tseng, C-T.Yang, "Scheduling of precedence constrained tasks on multiprocessor systems," *Proc. 1st Intl. Conf. on Algorithms and Architectures for Parallel Processing (ICAPP 95)*, Brisbane, Qld., Australia, April 19–21, 1995.

- [Chap96] Steve J. Chapin, "Distributed and multiprocessor scheduling," *ACM Computing Surveys*, vol. 28, no. 1, March 1996.
- [ScPi96] E.Di Sciascio, G.Piscitelli, "Performance evaluation of dynamic scheduling algorithms on a multiprocessor cluster," *Proc. 8th Mediterranean Electrotechnical Conference on Industrial Applications in Power Systems, Computer Science and Telecommunications (MELECON 96)*, Bari, Italy, May 13–16, 1996.
- [FPRu96] H.Franke, P.Pattnaik, L.Rudolph, "Gang scheduling for highly efficient distributed multiprocessor systems," *Proc. 6th Symp. on Frontiers of Massively Parallel Computation (Frontiers '96)*, Annapolis, Maryland, Oct. 27–31, 1996
- [ZHZh97] L.Zhang, J.Huang, Y.Zheng, "Scheduling algorithms for multiprocessor real-time systems," *Proc. Intl. Conf. on Information, Communications, and Signal Processing (ICICS'97)*, Singapore, September 9–12, 1997.
- [LiPa97] J-C.Liou, M.A.Palis, "A comparison of general approaches to multiprocessor scheduling," *Proc. 11th Intl. Parallel Processing Symp.*, Geneva, Switzerland, April 1-5, 1997.