

ECE497NC Final Report: RC Kernel Selection and Scheduling

Derek B. Gottlieb, Brian Greskamp, Richard B. Kujoth
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{dgottlie, greskamp, kujoth}@crhc.uiuc.edu

Abstract

Runtime Reconfiguration (RTR) allows reconfigurable architectures to adjust to application phased behavior at runtime. In this paper, we examine the performance of one such RTR system and the impact of system parameters such as reconfigurable array size and reconfiguration time. To gauge the effectiveness of our RTR algorithm, we compare its performance against an equivalent system that relies on static kernel scheduling where the compiler selects a set of kernels that will be mapped onto reconfigurable logic for the duration of an application. We demonstrate a speedup of up to 50% for a range of logic sizes over a static schedule when using a simple RTR algorithm for reasonably-sized reconfigurable coprocessors and suggest improvements to the algorithm that should widen this gap even further.

1 Introduction

Programmable-reconfigurable processors have the potential to exploit the performance advantages of reconfigurable logic while maintaining the programmable processor’s ability to execute arbitrarily large applications that would be prohibitive to implement in hardware [3] [8]. In order to take advantage of these systems, designers must confront the problem of partitioning an application between the programmable and reconfigurable resources. For this paper, the partitioning problem

decomposes into two subproblems: kernel selection and scheduling.

The partitioning problem is motivated by the fact that space in reconfigurable logic is limited. Since the entire application will not fit in hardware, the majority must execute in a conventional programmable processor that communicates with the reconfigurable logic. Generally, the sequential, memory-bound, or infrequently executed parts of the computation should execute on the processor while those that have high ILP and are frequently executed should run in reconfigurable logic. Other considerations in partitioning are the communication demands between the programmable and reconfigurable processors and the amount of time required to reconfigure the logic. The goal of kernel selection then is to generate configuration bitstreams for sections of the code (hereafter referred to as ‘kernels’) which will likely derive benefit from executing in reconfigurable logic. Scheduling deals with actually configuring some subset of these kernels in the reconfigurable logic either before the application begins execution or as it executes.

Kernel selection is usually done at compile time rather than run-time since producing hardware configuration bitstreams has traditionally been time-consuming. To this end, we detail a method for kernel selection based on hyperblock formation with a novel style of profile-directed compilation. Kernel scheduling may be performed either statically at compile-time or dynamically at run-time. While static scheduling may take advantage of profiling information, this profiling infor-

mation may depend heavily on the input data set for the application, resulting in poor performance in future executions with different input data. On the other hand, systems with run-time reconfiguration have the potential to automatically adapt to phased behavior and changes in the input data at the expense of additional complexity.

This paper seeks to explore some of the design parameters that should be considered when developing a programmable-reconfigurable processor with run-time reconfiguration support and attempts to quantify how RTR performance compares with a simpler statically scheduled system. Optimally, an RTR system should schedule onto hardware those segments of the application (kernels) that provide the greatest possible speedup at every point in time. However, the performance of RTR systems is limited by the efficacy of the kernel scheduling heuristic. The fact that future runtime behavior can never be predicted with 100% accuracy means that, at least theoretically, there will exist pathological cases where static scheduling outperforms any given online scheduling heuristic. We believe, however, that run-time kernel scheduling can be made to perform very well in practice. Since we have the choice of implementing the scheduler in hardware, software, or any combination thereof, many heuristics incurring different complexity and overheads are open for consideration.

In this paper, we examine the performance of one such RTR system, which uses a simple runtime scheduling algorithm that attempts to maximize speedup by swapping the precompiled hyperblock-kernels in and out of hardware at runtime. Using trace-based simulation along with rough performance estimates for the RC kernels, we calculate the speedup realized by this system and compare it with a system where the RC logic configuration is fixed before the program begins executing. We examine the impact of system parameters such as size of the reconfigurable array and time required to reconfigure, identifying key issues and tradeoffs in programmable-reconfigurable partitioning.

The rest of the paper describes the three tools used in our analysis. A profile-directed kernel selection algorithm using hyperblocks features in

Section 2. Section 3 describes the benefits of path profiling over traditional basic block and control-flow edge profiling and discusses the implementation details of our approach. An overview of the trace-driven simulator used to evaluate RTR scheduling appears in Section 4. We then discuss the results of our simulations in Section 5 and finally conclude.

2 Hyperblock Generation

Hyperblocks originated in the context of code scheduling for VLIW machines, where their purpose is to designate the most common paths through an inner loop. With predication, multiple speculative control flow paths can execute simultaneously. Thus the most common paths are executed speculatively on each iteration and if an uncommon path is taken, control flow exits the hyperblock and an external handler is invoked to perform the demanded computation. Since multiple exceptional paths are possible, the hyperblock can have several exits. However, each hyperblock has by definition only one entry point, usually the head of a loop. The hyperblock is a useful construct in mapping code to reconfigurable logic because excluding uncommon paths saves space in reconfigurable hardware. Compilers for reconfigurable logic have long exploited this fact; Garp uses a subtractive hyperblock formation scheme [4, 2] wherein the entire loop is initially considered and paths are removed until a figure of merit based on configuration size and critical path length is maximized.

In accordance with Garp's kernel formation strategy but in contrast with VLIW tradition, we have chosen to allow hyperblocks to encapsulate entire loop nests on the assumption that including outer loops should reduce the hardware invocation overhead. Each time control passes between the RC logic and the processor, a communications penalty is incurred transferring the live-in and live-out values for the kernel. It *may* therefore be desirable to place the entire loop nest in hardware. On the other hand, it may be best to include the inner loop only. To handle either eventuality, the compiler will generate all possible hyperblock

configurations, from the minimal inner-loop only to the entire nest, and runtime environment will be responsible for scheduling the best one into hardware.

2.1 Method

The input to the hyperblock generator is the path profile of Section 3, which gives the execution frequency for each acyclic program path. For each executed loop, there will be at least one path from the loop head to a loop exit. In the case of nested loops, the path for the outer loop includes the entry and exit nodes of the inner loop, but not the paths taken within the inner loop. For example, the paths in Table 1 are a valid profile for the code in Figure 1. The profile reflects that the outer loop executes five iterations and that the inner loop executes fifty iterations. The branch at node 6 is heavily weighted toward 8, and the exceptional exit 5 → 12 is never taken.

Frequency	Path
1	(1, 2, 11, 12)
5	(2, 3, 10, 11)
48	(3, 4, 6, 8, 9, 10)
2	(3, 4, 6, 7, 10)

Table 1. Example path profile for the code in Figure 1.

To build hyperblocks, we iterate over all loop headers in the program. For each loop header, a figure of merit is computed for each path starting at the header. The path with the highest merit is added to the hyperblock if it exceeds a pre-set minimum merit and the merits for all paths starting at the loop header (other than the one just added) are recomputed. This ‘calculate-merits’ then ‘add-best’ cycle iterates until the hyperblock reaches a pre-set size limit or until all paths exceeding the minimum merit have been added. Nested loops are treated as follows: Whenever a path includes a nested loop, the hyperblock for the nested loop is first built recursively then the merit for the path is calculated, taking into consideration all of the instructions added to the nested hyperblock. The

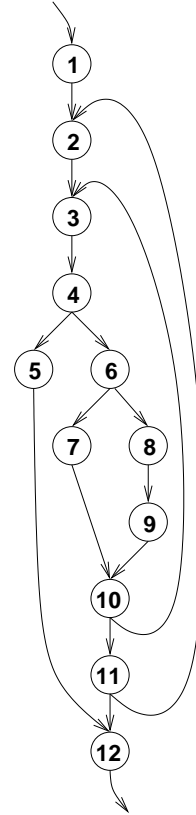


Figure 1. A doubly-nested loop with exceptional exit.

figure of merit for a path is the quotient of its execution frequency and the estimated area it requires in reconfigurable hardware. The area estimate is accumulated by simply examining the opcodes of each instruction in the path. For example, every `add` or `sub` opcode is estimated to take one ‘row’ of reconfigurable logic and a multiplication is estimated to take four rows. These numbers would have to be tuned for different reconfigurable substrates.

2.2 A Short Example

This section goes through the process of forming a hyperblock at basic block A in figure 2. The numbers to the right of each basic block are the blocks’ area estimates. Table 2 gives the path profile for the CFG. We will assume a minimum path merit of 2 and unlimited total area. Hyperblock

formation proceeds as follows.

Path	Frequency
<i>ABLNO</i>	1019
<i>ACFHKMNO</i>	227
<i>ACEHJMNO</i>	33
<i>ACFHJMNO</i>	5
<i>ACFHKO</i>	2
<i>BDIL</i>	1001
<i>BCL</i>	18

Table 2. Example path profiles for the CFG of figure 2.

The pass first examines loop header A. The first path from A (*ABLNO*) contains a nested loop at B. Therefore, we begin building a hyperblock at B. The first path from B is *BDIL*. Its merit is $101/(6 + 29 + 12 + 7)$ or 18. The other path from B is *BCL*, which has merit 0 and doesn't meet the minimum. With that, construction of the hyperblock at B is complete. It contains path *BDIL* only, for a total area of 54. Now we can resume examining path *ABLNO*. The area of this path (including the nested hyperblock at B) is $5+54+5+7$ or 71. Its merit is $1019/71$ or 14. This path won't actually be added until we examine the other paths from A to determine if there is one of higher merit. In fact, there is no such path and we find merit 2 for *ACFHKMNO* and 0 for *ACEHJMNO*, *ACFHJMNO*, and *ACFHKO*. Therefore, path *ABLNO* will be added next and the merits for other paths from A which have not yet been added will be re-calculated. This time, nodes A, N, and O will not count against the candidate paths' area since they have already been added. After merit recalculation, path *ACFHKMNO* receives merit 3 and all others remain at 0, so *ACFHKMNO* is added. At last, no other paths meet the minimum merit and the hyperblock at A is complete.

2.3 Suggested Improvements

The hyperblock formation algorithm is 'greedy', choosing the path of highest merit at each step. Obviously, this does not always lead to optimal

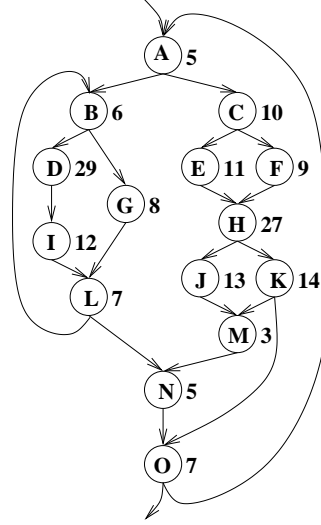


Figure 2. A simple loop nest from which to form hyperblocks. Execution frequencies for each path are in table 2.

path choices, but choosing the strictly optimal set of paths is NP-complete. While heuristics are indeed necessary, it is clear that the figure of merit (execution frequency divided by area) chosen above is not always the best criterion for path selection. Most importantly, it does not take into account the amount of ILP in any given path. With the current metric, a frequently executed path that contains a long dependency chain is favored over a less frequent, highly parallel one. In future versions of the algorithm, dependency height may be considered. Anyway, it is beneficial to calculate some parallelism estimate after the hyperblocks have been formed in order to provide the RTR simulator in Section 4 with approximate execution times for the hardware configurations.

Another weakness of the current algorithm is that it is intraprocedural, unable to generate kernels for loop nests with procedure calls. This limitation might be removed by automatic inlining.

2.4 Implementation

The hyperblock generation pass is implemented in the Machine SUIF [7] infrastructure. It operates in two distinct phases; the first phase exports information about the CFG loop structure to the path profiler and the second phase imports the path profiles from the simulator, invokes the hyperblock formation heuristic outlined above, and dumps information about the hyperblocks it generated. Specifically, the first phase marks loop headers, back edges, and exit arcs and dumps the information to a text file called `proc.brk` where `proc` is the name of a procedure. The starting and ending points of each edge are designated by label symbols, but the simulator expects them to be absolute numerical addresses. A PERL script called `lab2sym` reads the symbols from the application binary and translates the symbols to absolute addresses. The path profiles from the simulator are deposited in `bin.pro` where `bin` is the name of the binary. Of course the `.pro` file has numerical addresses where the compiler expects symbolic labels so the PERL script `addr2lab` performs the reverse translation. After the second phase completes, the compiler dumps a textual description of the hyperblocks it selected on `stdout`. For each hyperblock, it gives a starting label, one or more exit labels, and an area estimate. Future versions will also include an estimate of the hyperblock’s execution time in reconfigurable logic.

3 Path Profiling

In order to generate effective hyperblocks, some form of profiling is typically employed to help identify the heavily executed paths within a program. This is done by performing dynamic analysis on a running program to determine the execution frequencies for the different acyclic paths within the program. Profile information based solely on the execution frequencies of basic blocks and control-

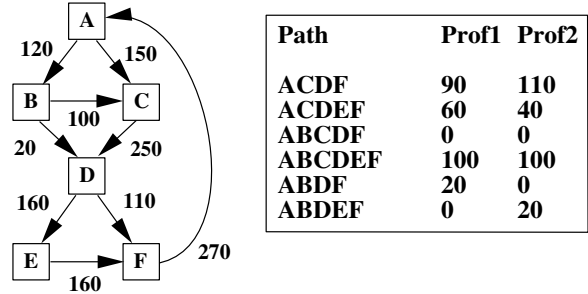


Figure 3. Profiling¹

flow edges is relatively inexpensive to produce, but fails to provide sufficient information to accurately predict the execution frequencies of overlapping paths. On the other hand, profiling the execution frequency of complete paths yields more accurate results at the expense of a manageable increase in complexity.

The distinction between edge and path profiling is illustrated by the control-flow graph (CFG) in Figure 3. In this figure, each edge in the CFG is labeled with its execution frequency for a given profiling run. A commonly used heuristic to select a heavily executed path relies on the most frequently executed edge out of a basic block. When considering block *A* in this figure, this heuristic would give preference to edge *A* → *C* because it executed 30 times more than edge *A* → *B*. Continuing on this basis, path *ACDEF* would be selected as the most common path in this graph.

However, it is possible for many different profiles to generate the same set of edge frequencies. The table accompanying Figure 3 lists the path frequency for two profiles that, despite different path frequencies, both result in the edge frequencies depicted in the CFG (*e.g.*, paths *ACDF* and *ACDEF* contribute to the total frequency of edge *A* → *C*). As a result, relying on edge frequencies alone can lead to sub-optimal path selection due to the effects of overlapping paths.

As mentioned previously, the most common path in this CFG is *ACDEF* according to a simple edge frequency-based heuristic. In profile 1,

¹Figure credit: Efficient Path Profiling by T. Ball and J. R. Larus

this path is only executed 60 times while paths *ABCDEF* and *ACDF* are executed 100 and 90 times respectively. Looking at the total execution behavior for this CFG, the path selected from edge frequencies accounts for only 22% of the overall execution. The situation is even worse in profile 2 since *ACDEF* accounts for only 15% of the overall execution. The accuracy of these path profiles are essential to the formation of effective hyperblocks.

3.1 Implementation

Traditionally, compilers perform path profiling by statically analyzing a program, assigning a unique index to each path, and then instrumenting the binary for the program with hooks that generate these indices based on the path being executed and updating an array of counters using that index at the end of an acyclic path. An efficient algorithm for implementing this is described in [1]. The instrumented program is then run and the compiler analyzes the resulting counter values to determine path frequency.

Since we are using *amalsim* [5] instead of actual hardware, we elected to extend the simulator to generate path profiling information instead of instrumenting the binary. Supporting path profiling in the simulator involves the development of a few key operations. The first operation is the actual generation of paths, which is easily accomplished as paths are represented by linked lists. When an instruction executes in an acyclic path, a new element is simply tacked onto the end of this list thereby maintaining correct sequential ordering. The second operation is the recording of these paths in a table when we've reached the end an acyclic path. This table is indexed using the address of the first instruction in the path, maintains a list of paths seen so far starting at this address and a counter for each of these paths. When a path is recorded in this table, it is first compared against the existing paths and either increments the counter if a match is found or adds this path to the list if this is the first time it's been executed.

To properly handle subroutines and nested loops, the current active paths are actually represented by a stack. When the program performs

a subroutine call or enters a loop body, it pushes the current path onto this stack and starts a new path at that point. When the program returns from a subroutine or completes a loop iteration, the current path is recorded in the table and the previous path is popped off the stack. In this manner, arbitrary levels of loop nesting and subroutine calls may be properly handled.

To facilitate stack management based on entering and exiting from loops, the compiler generates a *.brk* file that includes the location of loop boundaries based on information stored in the program's control-flow graph. Each entry in the *.brk* file consists of four fields: the starting address of the CFG edge, the end address of that edge, whether we need to push a new context, and how many contexts we need to pop. If both the third and fourth fields are equal to 1, then we are dealing with a back or loop edge and need to both record the current path through the loop and start a new path. When the simulator executes an instruction that is the starting address of a CFG edge in this file, it searches for the entry corresponding to its current branch target and performs the push/pop operations specified by the compiler. Using this information, it is possible for our path profiler to properly identify loops and handle nested loops.

3.2 Rationale

While this approach to path profiling is not as efficient as performing the necessary analysis in the compiler and instrumenting the binary, it may be more appropriate considering all program execution is performed using a simulator instead of actual hardware. Binaries instrumented with Ball-Larus path profiling incurs an average runtime overhead of roughly 45% over uninstrumented code for the SPECINT95 benchmark suite. While this is not a significant price to pay when running code natively on a processor, it would result in substantially longer runtimes for instrumented binaries running on a simulator. As an example, *amalsim* running on a 1.4GHz machine typically achieves a clock rate of roughly 100kHz, a difference of several orders of magnitude. This differen-

tial argues for extending the simulator to perform this analysis instead of instrumenting the binary, since code added to the simulator will be executed at the speed of the host machine which will be trivial compared to running the additional instrumentation code at the speed of the simulator. Initial results suggest this is a valid assumption as these additions to the simulator only reduce the simulator’s clock rate by a few percent.

4 RTR Simulator

We have designed a trace-based simulator, named *rtrsim*, that simulates the performance of a programmable-reconfigurable processor with various methods of reconfiguration. The goal of *rtrsim* is to estimate the execution time for an input program, using a specified processor configuration, determined by some set of command-line options as described below.

4.1 Implementation

rtrsim requires several inputs to execute a program. Inputs to the trace-based simulator include:

- A `.tra` file – a trace of the program execution on a single programmable processor
- A `.hbk` file – a list of hyperblocks w/hyperblock-specific info:
 - The initial PC of the hyperblock
 - An estimate of the hyperblock’s area in reconfigurable logic
 - A list of the possible exit PCs

The `.tra` file is generated using *amalsim*, a cycle-accurate simulator for amalgam, a clustered programmable-reconfigurable processor. *amalsim* is configured to use a single programmable cluster, similar to a small traditional microprocessor. The `.hbk` file is compiled from the results of generating the hyperblocks, as described earlier.

Rtrsim simulates a programmable-reconfigurable processor with partial reconfiguration in the reconfigurable array, and includes a configuration cache. The simulator

is very extensible, with several options for the processor architecture possible. Command-line options include:

- Dynamic/Static reconfiguration
- Sliding window size
- Reconfigurable logic size
- Reconfiguration latency
- Configuration cache size
- Configuration cache load time

4.1.1 Calculating Hyperblock Execution Time

Hyperblock execution time is estimated in the simulator by calculating the data dependence tree height. Memory operations, however, must execute sequentially. This approximation of the hyperblock execution time mimics the way that the hyperblock can be executed in reconfigurable logic, with control dependences being pushed further down in the control flow graph, and in a sense, executing all possible paths of the hyperblock.

By keeping memory operations in order, we are not giving any memory bandwidth or size advantages to the reconfigurable logic over the programmable processor.

4.1.2 Static Reconfiguration

If the simulator is set to do static reconfiguration, it decides which hyperblocks to insert into the reconfigurable array based on the profiling information, selecting hyperblocks with the best speedup per area ratio. While *rtrsim* iterates through the instructions listed in the `.tra` file, it checks each instruction’s PC versus the start PCs of each of the hyperblocks in reconfigurable logic. If the current instruction’s PC is the start of a hyperblock, the simulator skips ahead in the trace to the end of the hyperblock (which can be found by looking at the exit PCs of the hyperblock) and increments the running total of the execution time of the program by the estimate of the entire hyperblock’s execution time. If the current instruction is not

within a hyperblock that is included in the reconfigurable logic, then this instruction is considered to be executed on the programmable section of the processor, incrementing the program’s execution time by 1.

4.1.3 Dynamic Reconfiguration

If the simulator is set to do dynamic reconfiguration, execution of the program follows exactly as it does in the static reconfiguration case, except that hyperblocks are dynamically mapped to the reconfigurable logic, rather than a static set of hyperblocks loaded at the program start. The simulator keeps a sliding-window history of the hyperblocks and their execution time over a certain number of cycles, specified by the command-line options into *rtrsim*. Based on the history averages, the simulator swaps in hyperblocks that have been taking a large portion of the sliding-window history and swaps out those that have taken up a small portion of the sliding-window history. Execution of the program follows exactly as it does in the static reconfiguration case.

rtrsim implements partial reconfiguration of the reconfigurable logic array. This means that individual hyperblocks can be inserted, without having to reconfigure the entire logic array. Partial reconfiguration is also simplified so that if enough lines of the reconfigurable logic is available for a hyperblock to fit, then the hyperblock can be inserted, the simulator does not consider the possibility of the free space being segmented into many small portions of the logic array. Also, hyperblocks can be executed at the same time as another part of the reconfigurable logic is being reconfigured; this is a very optimistic assumption, but as our project is somewhat of a limit study, this will work.

4.1.4 Configuration Cache

We also included a configuration cache into the simulator. The configuration cache holds recently used configurations for loading into the reconfigurable logic quickly. Configurations can only be loaded into reconfigurable logic if they are currently in the configuration cache. If a configura-

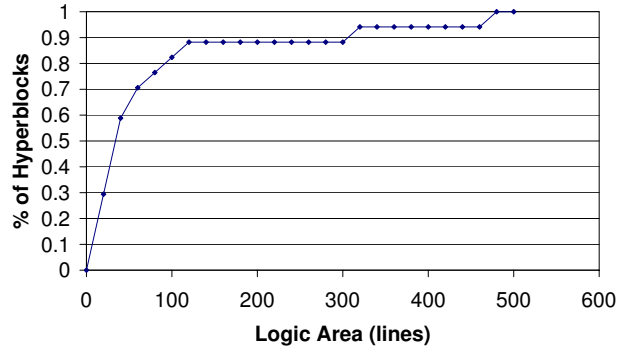


Figure 4. Cumulative graph of the sizes of the hyperblocks found in AES, Dither, DNA, FIR, MPEG, and TSP

tion is desired in reconfigurable logic, it first needs to be loaded into the configuration cache, and then loaded into the reconfigurable logic.

To eliminate the distraction of effects of the configuration cache in our results in the following section, we have disabled the configuration cache for the simulator used in this paper.

5 Results

With the infrastructure we have created, it is possible to investigate many areas of programmable-reconfigurable processors. We will limit our results in this paper to a few specific areas. The benchmarks that we will use to generate our results include: AES, Dither, DNA, FIR, MPEG, and the Traveling Salesman’s Problem (TSP).

5.1 Hyperblocks

We ran our hyperblock generating pass on the 6 benchmarks listed above. AES had 7 hyperblocks generated, MPEG had 4, DNA had 3, and FIR, Dither, and TSP had just 1 hyperblock generated. Figure 4 shows the sizes (in reconfigurable array lines, or rows of logic blocks) of the generated hyperblocks from all of the benchmarks. Nearly 90% of the hyperblocks generated required about 100 lines or less to implement in reconfigurable logic.

5.2 Dynamic/Static Reconfiguration

As discussed in the previous section, the main goal of *rtrsim* is to compare the execution times of a dynamically-reconfigurable processor against a statically-reconfigurable processor. In Figure 5, we see how several versions of a dynamically-reconfigured processor performs against a statically-reconfigured processor for the AES benchmark. The dynamically-reconfigured processor generally performs better than the statically-reconfigured processor, except in the case where the sliding window size is small (SW100 curve) and when the reconfigurable logic can hold all of the hyperblocks, in which case a statically-reconfigured processor would of course perform better.

The difference between the statically-reconfigured and the dynamically-reconfigured curves is very interesting between the range of 300 to 500 lines of reconfigurable logic. Two big hyperblocks (312 lines and 110 lines) generate significant speedups. In dynamic reconfiguration, the simulator is able to swap between the two, generating a significantly better speedup than the static reconfiguration, which could only load one of them. By swapping between them, dynamic reconfiguration raises the effective size of the reconfigurable logic by about 100 lines (enough to fit the second hyperblock). The relative sizes of the hyperblocks are somewhat irrelevant to the real implications of this, which is that when multiple hyperblocks of a program generate significant speedups, yet only a single one is able to fit in the array at a time, dynamic reconfiguration nets a huge performance benefit.

The SW100 curve is also very interesting. The performance of this dynamically-reconfigured processor actually decreases when the reconfigurable logic area is increased for short period. This is due to thrashing of the configurations in the reconfigurable array. Three hyperblocks are contesting for a spot in the reconfigurable array, with each of them evicting each other. It is not until all three hyperblocks can fit that the SW100 curve attains the performance of the other dynamically-reconfigurable processors with larger sliding win-

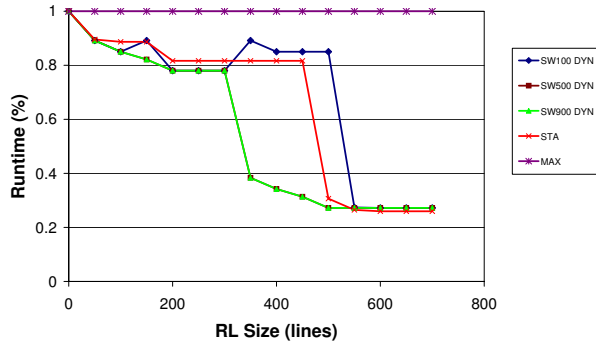


Figure 5. Comparison of dynamic and static reconfiguration on the AES benchmark

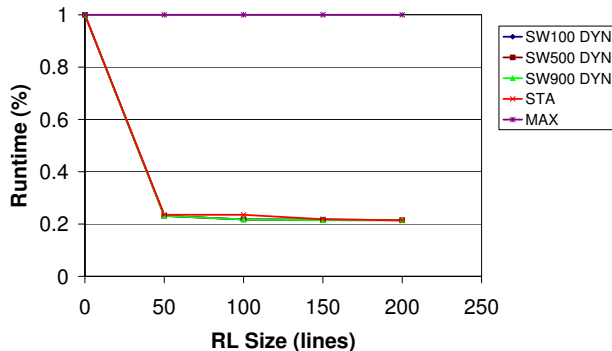


Figure 6. Comparison of dynamic and static reconfiguration on the MPEG benchmark

dows.

Figure 6 shows the same comparison on the MPEG benchmark. This graph is somewhat less interesting, since the statically and dynamically reconfigured cases perform nearly identically. This is due to the best hyperblocks fitting into a small number of lines of the reconfigurable array and the hyperblocks are executed numerous times, allowing the dynamically-reconfigured processor to only lose execution cycles compared to the statically-reconfigured processor on the first execution of the hyperblock.

Figure 7 is more or less a toy benchmark for this paper, since it only had a single hyperblock generated and this hyperblock is only executed a single time; thus, it is an even less interesting comparison, but does illustrate a drawback of

our method of dynamic reconfiguration. Since the statically-reconfigured processor loads configurations before it starts execution of the program, it had the single hyperblock in its reconfigurable logic before it was encountered in the program. In the dynamically-reconfigured processor, we only load configurations after program execution has begun and only then after we have seen a hyperblock in the sliding-window history. Because of this, we see that when the statically-reconfigured processor is able to fit the 478 line hyperblock into its reconfigurable logic, it is able to outperform the dynamically-reconfigured processor. Since the hyperblock is only executed once, the dynamically-reconfigured processor never takes advantage of using the hyperblock in reconfigurable logic.

Because FIR, Dither, and TSP all only generated a single hyperblock, we are only going to consider FIR for the results section, as the results for each would be similar and thus very trivial. The speedup will only occur when the single hyperblock is able to be mapped into reconfigurable logic and the difference between the static and dynamic cases would only be the number of times the hyperblock is executed. DNA, although three hyperblocks are generated, is similar, since a single hyperblock is where all of the significant speedup is obtained and is the only hyperblock executed more than once, so its results will also not be included.

We notice that the only interesting results are from the AES benchmark. This is due to the fact that it resulted in multiple hyperblocks that each gained speedup. This allows for more variation between the static and dynamic reconfiguration cases and thus more interesting results.

5.3 Notes on Dynamic Reconfiguration

Even our simple approach to selecting hyperblocks for dynamic insertion into the reconfigurable array leads to very nice performance increases over a statically-configured array. This shows promise for performing dynamic reconfiguration in a reconfigurable array. Something similar to our algorithm would not take much in the way of chip area, since it is essentially just a hash ta-

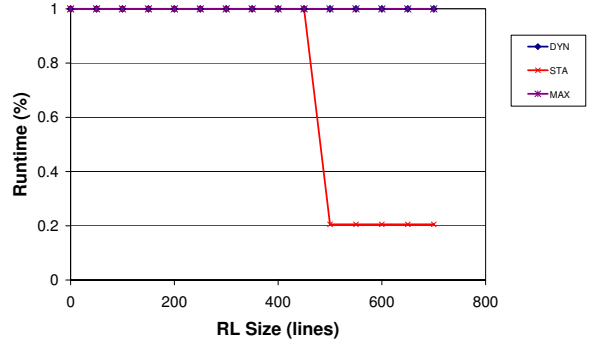


Figure 7. Comparison of dynamic and static reconfiguration on the FIR benchmark

[0][3]	- 100.000000%	(256/256)
[1][5]	- 100.000000%	(1/1)
[2][1]	- 0.390625%	(1/256)
[2][2]	- 99.609375%	(255/256)
[3][4]	- 100.000000%	(255/255)
[4][0]	- 100.000000%	(256/256)
[5][4]	- 1.923077%	(1/52)
[5][5]	- 98.076923%	(51/52)
[6][2]	- 100.000000%	(1/1)

Figure 8. Table of consecutively-executed hyperblocks for the AES benchmark

ble, keeping track of the recently executed hyperblocks, nothing more than a simple cache would need.

One interesting point that came out of obtaining these results was that we noticed that hyperblocks run in particular patterns depending on the benchmark. This is shown in Figure 8 and Figure 10, where a certain hyperblock almost *always* follows any given hyperblock. If this information was available to the algorithm for selecting insertion of hyperblocks, it could intelligently prefetch the likely next hyperblock. From these tables, the hyperblock flow graph has been obtained, seen in Figure 9 and Figure 11.

We could also imagine a scenario in which kernels are swapped in and out based on need. If a hyperblock is encountered in a program, it may be beneficial to swap that hyperblock into recon-

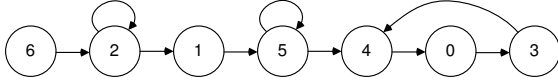


Figure 9. Illustration of the order of hyper-block execution in the AES benchmark

[0][1]	- 100.000000% (255/255)
[1][0]	- 6.250000% (256/4096)
[1][1]	- 93.750000% (3840/4096)
[2][1]	- 2.380952% (1/42)
[2][2]	- 97.619048% (41/42)
[3][2]	- 0.195312% (1/512)
[3][3]	- 99.804688% (511/512)

Figure 10. Table of consecutively-executed hyperblocks for the MPEG benchmark

figurable logic at that time if it is not currently in the reconfigurable logic.

5.4 Reconfiguration Time

An interesting, but somewhat unrelated result that we found involved varying the reconfiguration time of the reconfigurable array. As can be seen in Figure 12, varying the reconfiguration time of the reconfigurable logic, does not affect execution time much at all. The only spot where it is noticeable (210 cycle reconfiguration time in an array of size 150 lines) is due to some phase behavior, in which loading of the configuration took just long enough to miss the next execution of the hyperblock, but just in time to be evicted by an incoming hyperblock.

This result makes sense, however, since when a configuration is loaded, it is likely to stay in the reconfigurable logic for a while. The only way that

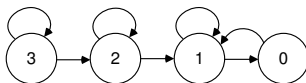


Figure 11. Illustration of the order of hyper-block execution in the MPEG benchmark

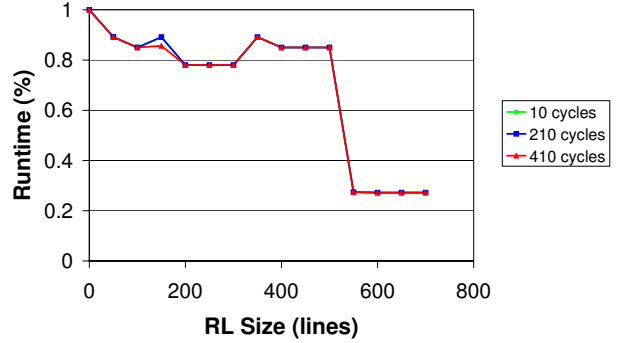


Figure 12. The effect of varying the reconfiguration time for different sizes of the reconfigurable array

the long reconfiguration time affects the execution time of the program is if the execution time of the hyperblock is so short that the configuration is still loading when the next (or next several) iterations of the hyperblock is executed.

This result may be due in part to our method of partial reconfiguration, since hyperblocks are still able to be executed in the reconfigurable logic while another configuration is loading.

6 Conclusions

This document described an RTR experiment in kernel selection and scheduling. The implementation consists of three major components. A Machine SUIF pass to process the path profiles and form hyperblocks appeared in Section 2. Section 3 outlined modifications to the Amalgam simulator to generate path profile information. Finally, the RTR simulator that allows for evaluation of the online scheduling algorithms was the subject of Section 4.

Section 5 covered simulation results, demonstrating that dynamic reconfiguration attains significant speedups over static reconfiguration. Dynamic reconfiguration gave 5% more speedup than static on the AES benchmark with a reasonably sized reconfigurable logic array size. For larger array sizes, dynamic reconfiguration claimed a nearly 50% performance improvement over static. These results illustrate how a program with sev-

eral hyperblocks of about the same size can benefit from dynamic reconfiguration — in the best case, effectively doubling the size of the reconfigurable logic by swapping hyperblocks in and out of the array. This result shows promise for dynamic reconfiguration, and with refinement of the scheduling and selection heuristics, we expect even larger performance gains.

References

- [1] Ball, T. and Larus, J. R.. “Efficient Path Profiling”, *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 46–57, December 1996.
- [2] Callahan, T. J. and Wawrzynek, J.. “Instruction-Level Parallelism for Reconfigurable Computing”, *International Workshop on Field Programmable Logic*, September 1998.
- [3] Callahan, T. J., Hauser, J. R., and Wawrzynek, J. “The Garp Architecture and C Compiler”, *IEEE Computer*, vol. 33, pp. 62–69, Apr. 2000.
- [4] Callahan, T. J.. “Kernel Formation in Garpcc”, *FCCM 2003*.
- [5] Gottlieb, D. B., Cook, J. J., Walstrom, J. D., Ferrera, S., Wang, C.-W., and Carter, N. P.. “Clustered programmable-reconfigurable processors”, *Proceedings of the IEEE International Conference on Field Programmable Technology*, pp. 134–141, December 2002.
- [6] Mahlke, S. A., Lin, D. C., Chen, W. Y., Hank, R. E., and Bringmann, R. A.. “Effective Compiler Support for Predicated Execution Using the Hyperblock”, *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 45–54, December 1992.
- [7] Smith, M. D.. “Extending SUIF for Machine-dependent Optimizations”, *Proc. First SUIF Compiler Workshop*, Stanford, CA, pp. 14–25, January 1996.
- [8] Ye, Z. A., Moshovos, A., Hauck, S., and Banerjee, P., “CHIMAERA: A High-Performance Architecture With a Tightly-Coupled Reconfigurable Unit,” *International Symposium on Computer Architecture*, pp. 225–35.