COMPILATION AND MANAGEMENT OF PHASE-OPTIMIZED RECONFIGURABLE SYSTEMS

Henry Styles and Wayne Luk

Department of Computing, Imperial College,
180 Queen's Gate, London, England
email:{hes2,wl}@doc.ic.ac.uk

ABSTRACT

A program phase is an interval over which the working set of the program remains more or less constant. This paper presents a dynamic optimization scheme which uses program phase information to optimize designs for reconfigurable computing. We present a mathematical formulation of the optimization problem and propose a solution which comprises of : (1) A hardware compilation scheme for generating configurations that are specialized for different phases of execution. (2) A runtime system which manages interchange of these configurations to maintain specialization between phase transitions. We report experimental results for Xilinx Virtex FPGAs involving OpenGL SPECview -perf benchmarks and demonstrate 95.39% speedup over an optimized uniform rate static design and 11.13% speedup over an optimized multi-initiation interval static design. We present a framework for *a posteriori* performance analysis and architectural exploration with which we (a) establish a performance upper bound under perfect phase optimization, (b) investigate sensitivity to reconfiguration time, (c) examine the quality of the proposed algorithm for phase-detection. The optimization is shown to be surprisingly insensitive to increased reconfiguration time. Faster reconfiguration yields limited benefits and performance improvements are possible upto 1 second reconfiguration time.

1. INTRODUCTION

Since the 1960s [1] it has been known that a broad set of programs exhibit phase behavior. Any program which adheres to the Phase Transition Model [2] has predicable program memory access patterns. Program phase is one of the basic principles which underpins cache design and branch prediction. Recently, microprocessors with reconfigurable cache configurations have been proposed [3] which include an explicit phase prediction model to specialize the cache configuration at runtime in response to phase change. This paper explores using phase behavior to optimize the mapping of computer programs to reconfigurable architectures such as FPGAs.

2. OPTIMIZATION FORMULATION AND EXISTING APPROACHES

Phase-optimization consists of generation of phase-optimized configurations and management of reconfiguration.

Configurations can be generated at runtime or offline. In microprocessors with multi-configuration caches [3] the different phase-optimized cache configurations are designed by hand offline. Several software [4] and hardware [5] environments have been reported which specialize designs at runtime. Offline generation may allow for greater specialization whilst runtime generation requires less state.

Configurations must be interchanged at runtime to maintain phase-specialization. This task is modeled using a trellis graph (Fig. 1):

1. Let t_p be the number of computation steps in a program execution.
2. Let c be the number of phase-optimized configurations.
3. $T \in \Re^{c \times t_p}$ stores trellis node weights. $T_{i,j}$ is the cost of configuration i for step j.
4. $R \in \Re^{c \times c}$ holds edge weights. $R_{i,j}$ is the reconfiguration time between configurations i and j.

A reconfiguration schedule is represented by $S \in N^{t_p}$, where S_i is the index of the configuration used at computation step i. The cost of S is its path length $length(S) = \sum T_{i,S_i} + \sum R_{S_i,S_{i-1}}$. The optimal reconfiguration schedule is the shortest-path through the trellis S_{opt} where $length(S_{opt})$ is minimal. Given a complete execution trellis, S_{opt} can be computed by simplified Dijkstra's Shortest Path [6]. The reconfiguration manager must compute an approximation of these operations at runtime. This consists of the following tasks :

1. Monitor the working set. At each $t_{current}$, a configuration independent measure of state called the *working set signature* is recorded. A windowed *working set history* of length w, $W_{past} \in sig^w$ is stored. In multi-configuration cache microprocessors [3], the

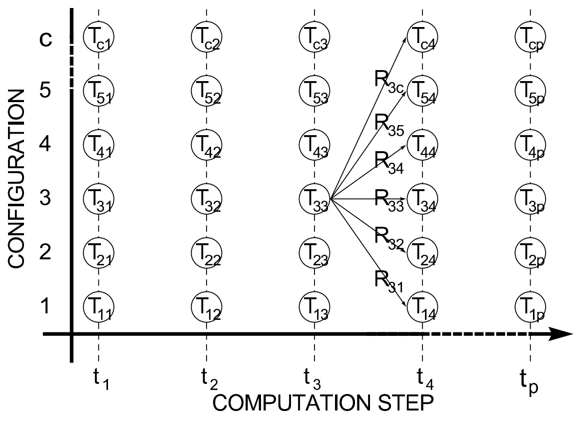
CONFIGURATION
COMPUTATION STEP
c
5
4
3
2
1
t_1
t_2
t_3
t_4
t_p
T_{c1} T_{c2} T_{c3} T_{c4} T_{cp}
T_{51} T_{52} T_{53} T_{54} T_{5p}
T_{41} T_{42} T_{43} T_{44} T_{4p}
T_{31} T_{32} T_{33} T_{34} T_{3p}
T_{21} T_{22} T_{23} T_{24} T_{2p}
T_{11} T_{12} T_{13} T_{14} T_{1p}
R_{3c}
R_{35}
R_{34}
R_{33}
R_{32}
R_{31}

Fig. 1. Trellis representation of execution history. $T_{i,j}$ is the cost of configuration i for step j. $R_{i,j}$ is the reconfiguration time between configurations i and j.

working set signature consists of a lossily compressed histogram of program counter values over time.
2. Working set sequence prediction. The future working set sequence $W_{future} \in sig^w$ is predicted from W_{past}.
3. Evaluate cost of alternative configurations. The future trellis window $T_{future} \in \Re^{c \times w}$ is determined from W_{future}.
4. Reconfiguration scheduling. The shortest path over T_{future} is estimated.
5. Invoke reconfiguration. The reconfiguration schedule is implemented by invoking reconfiguration at the specified time-steps.

In existing systems (3) is achieved by *tuning* [3] or *modeling*. A *tuning sequence* consists of systematically trying each of a number of configurations and measuring the performance of each. Tasks (2) and (4) are typically [3] bundled together in a combined algorithm.

3. PROPOSED SYSTEM

In our system, designs are specialized for different phases by optimizing resource allocation between different program branches. The optimal resource allocation for a phase is a function of program branch probabilities [7]. We define a program phase as an interval over which the branch probabilities of a program remain more or less constant. The proposed system consists of :

Generation of phase-optimized configurations We compile a single high level program into a spectrum of phase-optimized FPGA configurations offline. Our compilation scheme [7] [8] combines coarse grain asynchronous and fine grain synchronous pipelines and allows different program branches to operate at different initiation intervals. For a design of n basic blocks, parameter $b \in N^n$ sets the initiation interval of each block. b_i is the cycles per result of block i. The spectrum of configurations covers a subset of the Cartesian product of possible parameterizations, culled by applying flow heuristics. A parameterization is culled if it contains a sub-graph with

1. **Downstream slack** The sustainable output rate is greater than the sum of the maximum input rates.
2. **Upstream blocking** The sum of the input rates is greater than maximum sustainable output rate.

Management of reconfiguration Our reconfiguration manager mixes hardware and software. Monitoring of the working set is conducted in hardware with all other activities in software.

Monitor the working set We define the working set signature to be the set of branch probabilities over a finite execution window. The working set signature is recorded by profiling counters in hardware. Each BRANCH node contains two counters which record the number of branches and the total number of TRUE branches. At the end of a 10 million input-sample execution window the signature is fetched by software.

Evaluate cost of alternative configurations A simple steady state $M/M/1/\infty/FCFS$ queuing network model is compiled for each parameterization. In [7] we demonstrated that this model is both accurate and fast. The input parameters for each configuration are :

1. Initiation interval parameterization vector b.
2. The branch probabilities. Routing matrix $Q \in \Re^{n \times n}$, where element Q_{ij} is the steady state probability that a job completing basic block i branches to block j.

The traffic equations (eq. 1) are solved, subject to utilization constraints (eq. 2), to determine overall performance : the maximum sustainable input rate to block one γ_1.

$$\vec{\lambda}(I - Q) = \vec{\gamma} \qquad (1)$$
$$\lambda_i b_i \leq 1 \quad i = 1, 2, .., N \qquad (2)$$

We propose a partial evaluation scheme to minimize calculations at runtime. A symbolic solution is generated offline and at runtime Q is substituted in. This requires that only two sets of N linear equations need be evaluated at runtime. It has been suggested that if (eq. 1) is ill-conditioned, the partial evaluation should be abandoned and a full numeric solution should be computed at runtime.

Working set prediction, reconfiguration scheduling and invoke reconfiguration We propose a very simple combined algorithm which is based on a 1-step history. The algorithm reconfigures the device to the highest performance configuration over the previous execution window as determined by

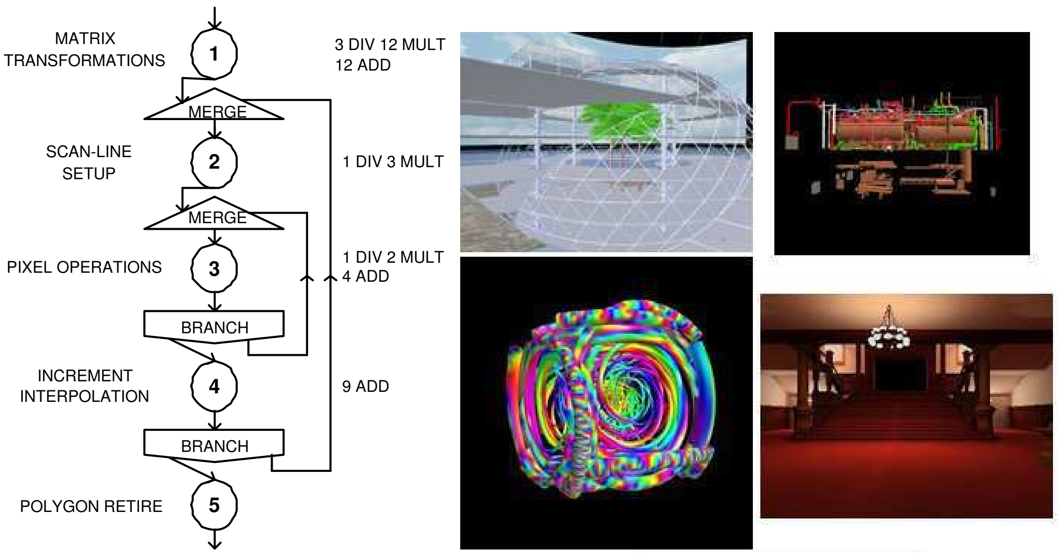

Fig. 2. Left pane shows the top level dataflow graph for Mesa3D implementation showing basic block activities and resource usage. Right pane shows the SPECviewperf benchmarks used in experiments. Images are taken from SPECviewperf web site. Panes in clockwise order from upper left. Test 1 : SPECapc 3D Studio Max 3.1, Test 4 : Intergraph Designreview, Test 6 : Discreet Lightscape, Test 5 : IBM Data Explorer.

the queueing network model. No reconfiguration occurs if the fastest configuration is active.

4. PERFORMANCE ANALYSIS: SPECVIEWPERF

OpenGL is an industry standard API for real-time rendering. We implement parts of the OpenGL-like Mesa3D [9] graphics library. The top-level dataflow graph is shown in the left pane of Fig. 2. There are five basic blocks with two feedback loop carry dependent loops. We use eight of the SPECviewperf [10] SPEC benchmarks for OpenGL. A combined benchmark is also examined which runs all benchmarks in sequence. All benchmarks were run at 320x200 resolution, 32-bit colour.

All experiments begin with compilation and synthesis of the phase-optimized configurations for Xilinx Virtex-E XCV1000-E. Our arithmetic library comes from Xilinx Core GENERATOR and supports initiation intervals 1, 2, 4, 8, 16 and 32 cycles per result. The compiler generates 56 different parameterizations for b, of which 54 fit the XCV1000-E.

4.1. Performance upper bound

For an arbitrary program and dataset, phase-optimization will deliver a theoretical maximum performance improvement when :

1. *Reconfiguration management overhead* is zero.
2. *Reconfiguration time* is zero.
3. *Detection efficiency* is one, such that the reconfiguration schedule is optimal.

Clearly, no real-world system will share these properties. However, analyzing performance under these conditions is useful as it defines a performance upper bound, below which all real-world phase-optimizing systems exist. Upper bound performance U_1 is determined by *a posteriori* analysis of the execution trellis. A trellis is computed for each benchmark using cycle-accurate simulation. The optimal reconfiguration schedule is then computed by shortest path. Two control experiments are reported. Control C_1 is the fastest single design with uniform rate for all blocks. Control C_2 is the fastest single design with multiple initiation intervals.

Table. 1 shows the results for these experiments. A timing constraint of 90MHz clock rate is met by all 54 designs in the spectrum of phase-optimized FPGA configurations. In total, 66 reconfigurations are made over the course of the nine benchmarks. Dynamic reconfiguration is present in four of the nine optimal reconfiguration schedules. Fig. 3 illustrates the optimal reconfiguration schedule for the combined benchmark.

The variance of speedup results for different datasets is significant and shows that phase optimization is a data-dependent optimization. Phase-optimization in only beneficial if the underlying assumption of phase-phase-behavior holds true. The flat sections of Fig. 3 illustrate that there is little phase behavior in benchmarks 1,3,4,5 and 7. The optimal reconfiguration schedules for these benchmarks there-

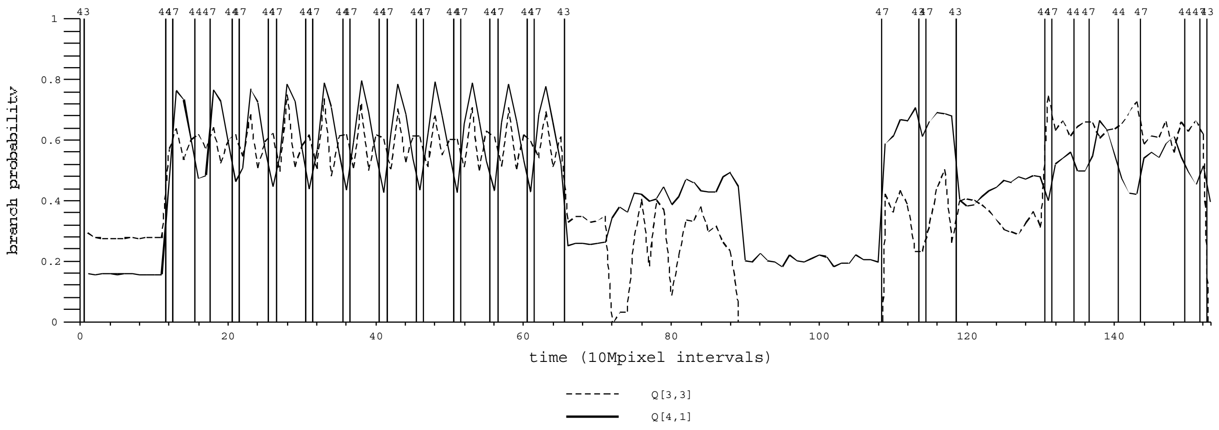

Fig. 3. Optimal reconfiguration schedule for upper bound performance measure, SPECviewperf benchmark 9. Lines Q[3,3] and Q[4,1] show the branch probabilities of the inner and outer loop over time. Design 43 corresponds to b parameterization of [4,2,2,1,1]. Design 44 corresponds to b parameterization of [4,4,1,1,1]. Design 47 is b parameterization of [8,1,1,1,1]. For SPECviewperf benchmark 8, the optimal reconfiguration schedule U_1 contains 36 reconfigurations.

		Configs		Time (ms)	Relative speedup of U_1 (%) over	
Test	Rf	U_1	C_2	U_1	C_1	C_2
1	0	43	43	2976	64.24	0
2	21	44,47	44	7996	200.15	2.784
3	0	43	43	1335	99.65	0
4	0	43	43	4202	90.47	0
5	0	43	43	4222	100.00	0
6	2	43,47	47	1907	133.00	2.205
7	0	43	43	2666	100.00	0
8	7	44,47	44	3163	209.08	2.753
9	36	43,44,47	44	36154	105.21	16.72

Table 1. Upper bound speedup for SPECviewperf benchmarks, XCV1000-E at 90 MHz. Rf shows the number of reconfigurations configuration schedule C_1. The *Configs* columns show the names of configurations used in each schedule. Configuration 43 is initialization interval b parameterization [4,2,2,1,1], 47 is [8,1,1,1,1], 44 is [4,4,1,1,1] and 46 is [4,4,4,1,1]. Schedule C_1 uses configuration 46 in all tests. The remaining columns show the percentage speedup of schedule U_1 over C_1 and C_2.

fore contain zero reconfigurations.

Benchmarks 2,6,8,9, exhibit more classical phase transition behavior. For these benchmarks, branching behavior oscillates between stable phases. The probabilities are sufficiently different and phases are of sufficient length to enable optimization by swapping between phase-optimized configurations. In benchmark 2, the upper bound optimal reconfiguration schedule oscillates between parameterization $b = [4, 4, 1, 1, 1]$ and $b = [8, 1, 1, 1, 1]$ through 21 reconfigurations. The upper bound speedup over the best single configuration is 2.784% with 200.15% speedup over the best single uniform rate configuration. In benchmark 8 the schedule includes 7 reconfigurations and the upper bound speedup over a single configuration scheme is 2.205%. A 133% speedup is achieved over the best uniform rate design. For benchmark 9, the important combined benchmark, the upper bound optimal reconfiguration schedule oscillates between parameterization $b = [4, 2, 2, 1, 1]$, $b = [4, 4, 1, 1, 1]$ and $b = [8, 1, 1, 1, 1]$. It consists of 36 reconfigurations and the upper bound speedup over the best multi-initiation interval design is 16.72%. A 105.21% speedup is achieved over the best uniform rate design.

Why is there little speedup over the best single multi-initiation interval design ? Firstly, there is a lack of phase-transition behavior in the dataset. In general the SPECviewperf benchmarks do not exhibit the classical behavior of the phase-transition execution model. If an application could be found with longer or more varied phases of execution, more reconfigurations would occur in the optimal reconfiguration schedule and a greater theoretical upper bound on performance improvement would be achieved. Secondly there is a lack of fine grain control on design specialization. Our arithmetic library is restricted to initialization intervals of powers of two. As a result there are large intervals of branch probability over which the same design parameterization is optimal. Greater flexibility, for example arbitrary integer initialization intervals, would permit finer grain specialization and encourage more frequent reconfiguration.

4.2. Architectural exploration: sensitivity to reconfiguration time

This section explores the sensitivity of the optimal reconfiguration schedule to increased reconfiguration time. We begin by constructing the complete execution trellis for each benchmark. The *a posteriori* shortest path is then computed with parameterized reconfiguration overhead edge costs R. Performance is analyzed over the interval of zero-cost reconfiguration (U_1) up to 2 seconds per reconfiguration.

$R_{i,j}$ (ms)	Test 2	Test 6	Test 8	Test 9
0	2.78 / 31	2.20 / 2	2.75 / 7	16.72 / 36
1	2.51 / 21	2.10 / 2	2.55 / 5	16.60 / 34
2	2.24 / 21	1.99 / 2	2.38 / 5	16.50 / 34
3	1.71 / 21	1.77 / 2	2.06 / 5	16.27 / 34
4	0.722 / 17	1.35 /2	1.51 / 4	15.86 / 29
16.4688	0 / 0	0.467 / 2	0.66 / 3	15.37 / 11
32	0 / 0	0 / 0	0 / 0	14.97 / 6
64	0 / 0	0 / 0	0 / 0	14.36 / 6
128	0 / 0	0 / 0	0 / 0	13.16 / 6
256	0 / 0	0 / 0	0 / 0	11.55 / 4
512	0 / 0	0 / 0	0 / 0	9.06 / 3
1024	0 / 0	0 / 0	0 / 0	4.88 / 3
2048	0 / 0	0 / 0	0 / 0	0 / 0

Table 2. Sensitivity of runtime phase optimization to increased reconfiguration overhead for SPECviewperf benchmarks 2,6,8 and 9. $R_{i,j}$ is the reconfiguration time in milliseconds. Remaining columns show the % speedup of U_1 over C_2 and the number of reconfigurations in U_1.

Table 2 shows optimal reconfiguration schedule performance degradation with increased reconfiguration overhead. Tolerance to high reconfiguration overhead is only possible in benchmarks which exhibit sufficient phase-transition behavior. SPECviewperf benchmarks 2, 6, and 8 exhibit limited phase behavior and are sensitive to increased reconfiguration time. Combined benchmark 9 exhibits greater phase behavior. The results for the combined benchmark show :

1. Where applicable, phase-optimization is well suited to existing architectures and complete device reconfiguration in the range of 10-50ms. The XCV1000-E (16.466 ms) suffers only 1.35% loss of only speedup compared to the zero-reconfiguration time optimal schedule.
2. Where applicable, phase-optimization is surprisingly insensitive to increased reconfiguration time. Performance improvements are still possible at 1 second reconfiguration time.
3. There is surprisingly little benefit in faster reconfiguration. Techniques such as partial reconfiguration or coarse grain reconfiguration would be ineffective.

4.3. Prototyping board experiments

This section describes experiments using the RC1000–PP board. All designs run at the maximum memory clock rate 25MHz. Four control experiments were conducted using techniques described in Section 4.1.

C_1 The fastest uniform rate single design.
C_2 The fastest multiple initiation interval single design.
C_3 The optimal reconfiguration schedule using designs with multiple initiation intervals. Zero reconfiguration time.
C_4 The optimal reconfiguration schedule using designs involving multiple initiation intervals. Reconfiguration time 16.4688ms for the XCV1000-E on the RC1000-PP.
I is the full phase-optimization system. Tables 3 and 4 show performance results.

	Rf			Configurations				
Test	C_3	C_4	I	C_2	C_3	C_4	I	meand
1	0	0	1	43	43	43	46,43	0
2	21	21	22	47	44,47	44,47	46,44,47	0.476
3	0	0	1	43	43	43	46,43	0
4	0	0	1	43	43	43	46,43	0
5	0	0	1	43	43	43	46,43	0
6	2	2	3	47	43,47	43,47	46,43,47	0
7	0	0	1	43	43	43	46,43	0.4
8	7	5	7	47	44,47	44,47	46,44,47	0.4
9	36	34	36	43	43,44,47	43,44,47	46,43,44,47	0.588

Table 3. SPECviewperf OpenGL benchmarks for RC1000–PP. *Rf* is the number of reconfigurations in the schedule. See Table 1 for explanation of *Configurations* column. Experiment C_1 uses configuration 46 in all tests. The *meand* column shows the mean time between reconfigurations in C_4 and I.

	Time (ms)	Relative speedup of I (%) over			
Test	I	C_1	C_2	C_3	C_4
1	11381	54.86	-5.71	-5.71	-5.70
2	32639	167.68	-8.33	-10.81	-10.69
3	5618	71.36	-14.17	-14.17	-14.12
4	15701	83.71	-3.55	-3.55	-3.54
5	16016	90.00	-5.00	-5.00	-4.99
6	7978	101.79	-11.48	-13.39	-13.31
7	10416	84.62	-7.69	-7.69	-7.68
8	12886	175.61	-8.37	-10.82	-10.82
9	137292	95.39	11.13	-4.79	-4.77

Table 4. SPECviewperf OpenGL benchmarks running on RC1000–PP. Table shows the percentage speedup of the implemented system I over the control experiments.

Table 4 shows overall experimental speedup. Column three shows the speedup in percentage of the experimental runtime phase optimizing system I versus the best single configuration with uniform initiation interval. Speedup ranges from a 54.86% speedup for benchmark one, to a 175.61% speedup for benchmark eight. For the combined benchmark a speedup of 95.39% is encountered. Column four shows that for the combined benchmark, an 11.13% speed improvement is made over the best possible single configuration. Column five and six show that I is 4.77% slower than the optimal possible reconfiguration strategy for the XCV1000-E.

4.4. Quality of phase-detection scheme

Table 3 lists the configurations used and the number of reconfigurations during execution. The results indicate that the experimental reconfiguration management system performs well. The number of reconfigurations used in I correlates well with the optimal reconfiguration schedule for XCV1000-E, control study C_4. I also uses the same configurations as C_4 with configuration 6 used on startup.

The final column of Table 3 shows the mean phase change miss distance of experimental schedule I compared to C_4. Our phase detection algorithm exhibits high *detection efficiency*: it schedules each reconfiguration on average less than half a profiling sample away from the optimum reconfiguration schedule.

5. CONCLUSIONS

The key contribution of this work is a system of phase- optimization which comprises (1) A hardware compilation scheme for generating configurations that are specialized for different phases of execution. (2) A runtime system which manages interchange of these configurations to maintain specialization across phase transitions. We provide an experimental implementation for Xilinx Virtex FPGAs and demonstrate 95.39% speedup over an optimized uniform rate static design and 11.13% speedup over an optimized multi-initiation interval static design.

We characterize the zero-reconfiguration time upper bound on performance and explore the sensitivity of the proposed system to increased reconfiguration time. The upper bound for XCV1000-E at 90MHz is 16.72% over the best possible single configuration. Performance degrades gracefully as reconfiguration time is increased. The optimization is shown to be beneficial in the 10-50ms reconfiguration time region exhibited by modern FPGAs and is surprisingly insensitive to increased reconfiguration time. Performance improvements are possible upto 1 second reconfiguration time and there is little benefit in faster reconfiguration.

We analyze the quality of the proposed reconfiguration management system. The runtime system is shown to be extremely lightweight : only two sets of linear equations need be evaluated at each timestep. The overall performance of the system is only 4.77% slower than the optimal possible reconfiguration strategy for the XCV1000-E and the phase detection algorithm is shown to exhibit a very high detection efficiency.

There are several possible directions for future work. The most pressing requirement is to build a flexible arithmetic library which targets Virtex IV to address issues raised in Section 4.1. Our compilation scheme generates a globally asynchronous locally synchronous design. Greater specialization will be sought in multiple-clock domain configurations, in effect enabling more flexible selection of initialization intervals. There is also significant scope for improving management of reconfiguration. Our queuing network model would be improved by attempting to model burstyness. More sophisticated phase change detection algorithm algorithms such as the *Signature-Based Reconfiguration Algorithm* or *Rochester Algorithm* [3] will also be investigated. Finally, modern FPGAs are capable of self-reconfiguration [11], inviting the possibility of phase-optimizing system-on-chip.

6. REFERENCES

[1] P. Denning, "The working set model for program behavior," in *Proceedings of the ACM symposium on Operating System Principles*, 1967, pp. 15.1–15.12.

[2] A. Batson and A. Madison, "Measurements of major locality phases in symbolic reference strings," in *Proceedings of the International Symposium on Computer Performance, Modeling, Measurement and Evaluation*, 1976, pp. 75–84.

[3] A. S. Dhodapkar and J. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *Proceedings of the 29th IEEE/ACM International Symposium on Computer Architecture*, 2002, pp. 233–244.

[4] G.-Y. L. M. Cierniak and J. Stichnoth, "Practicing judo : Java under dynamic optimization," in *SIGPLAN*. ACM Press, 2000.

[5] S. McMillan and S. Guccione, "Partial run-time reconfiguration using jrtr," in *Field Programmable Logic and Applications*. Springer, 2000.

[6] E. Dijkstra, "A note on two problems in connection with graphs," *Numeriche Mathe*, vol. 1, pp. 269–271, 1959.

[7] H. Styles and W. Luk, "Branch optimisation techniques for hardware compilation," in *Field Programmable Logic and Applications*. Springer, 2003, pp. 324–333.

[8] H. Styles, D. Thomas, and W. Luk, "Pipelining designs with loop-carried dependencies," in *International Conference on Field Programmable Technology*. IEEE Computer Society Press, 2004.

[9] B. Paul, "The mesa 3d graphics library," http://www.mesa3d.org/.

[10] "Specviewperf 7.1," http://www.specbench.org.

[11] B. Blodget, P. Roxby, E. Keller, S. McMillan, and P. Sundararajan, "A self-reconfiguring platform," in *Field Programmable Logic and Applications*. Springer, 2003, pp. 565–574.